Software Product and Process Quality Improvement Using Formal Methods

Dissertation

zur Erlangung des akademischen Grades
(doctor rerum naturalium, Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
Humboldt-Universität zu Berlin

von

Herr Satish Mishra

Tag der mündlichen Prüfung 10.07.2014

Contents

List of Figures

List of Tables

Abstract

Currently, software systems are incorporating more and more functionalities, which lead to an ever increasing complexity. The growing complexity is a major challenge for the development of software systems. A systematic approach is required for the development of such software systems. Formal methods have shown significant benefits for the development of such software systems. Particularly, formal methods are well known for requirement analysis, correctness preserving techniques, verification and validation aspects in software development. These development aspects are viewed as product quality; generally concerned with product features. On the other hand there is process quality which is a collection of best practices. The best practices have been proposed by various process models such as ISO, Six-Sigma, CMMI. These process models have benefited the development of predictable software systems in organizations. Generally, product and process quality goals are achieved independently for the development of software systems. In this research, we propose a unique approach to integrate these two aspects of software system quality improvements.

This research is based on the foundations of formal methods such as software specification, refinement, enhancement, testing and the process improvement model CMMI. We extend the existing foundations of formal methods as follows. In our consideration a software specification is considered as a consolidation of observable and internal behaviors. First, an abstract description of a software system is written as observable behaviors. Further, design details are added as internal behaviors. The syntax and semantics of the proposed specification formalism are described with an integrated structural and behavioral specification language CSP-CASL. The approach of software refinement is extended as constructive refinement. It allows existing refinement techniques on observable behaviors. The internal behaviors are refined by describing software design decisions as addition of internal behaviors. The complete refinement framework is elaborated for a CSP-CASL specification formalism.

Further, we formally define a concept of software enhancement within the framework of our proposed specification technique. In this consideration, a software enhancement is a process of adding new functional or performance requirements to the existing software system by semantically preserving its existing functionalities. The complete understanding of the software enhancement is described with the process algebraic and algebraic specification language, CSP-CASL. Subsequently, the syntax of CSP-CASL is extended to incorporate the failures of a specifying system. This extended CSP-CASL syntax is used to generate positive as well as negative test cases. This approach of test generation guarantees the expected as well as restricted properties of software system are in test cases. Further, testing terminologies are described for the CSP-CASL specification and explored in software refinement and enhancement. The given definitions are the fundamental aspects in the proposal of our software product and process quality framework.

The proposed formalisms and the other properties of formal methods are used to propose a framework of CMMI process model compliance. The core aspects of the CMMI process model are the process areas. A process area is a collection of best practices in a selected area. The CMMI compliance grading scheme is developed to evaluate the level of compliance with formal method based software development. A compliance algorithm is proposed to evaluate the process model through the evaluation of its components. The CMMI process areas are evaluated with a proposed algorithm. The compliance evaluation result is presented in the thesis. The complete framework is supported with a developed tool. This tool allows us to practically support our theoretical concepts. As a proof of concept, we explore our proposed framework for a medical instrument development and maintenance.

In this thesis, the understanding of formal methods applicability is extended to the organizational process model, CMMI. The complete framework is presented for a formal specification language, CSP-CASL and process model, CMMI. However, similar result can be achieved with other formal methods for the compliance of other process models. This research is a starting point of process model compliance with formal methods. This has significant potential to automate the achievement of process and product quality goals of software systems.

Zusammenfassung

Softwaresysteme vereinen heutzutage mehr und mehr Funktionen, was zu einer stetig steigenden Komplexität der Anwendungen führt. Diese wachsende Komplexität stellt eine der Hauptherausforderungen für Entwicklung, Test und Wartung von Softwaresystemen dar. Formale Methoden haben gezeigt, dass sie bei Entwicklung, Test und Wartung komplex Software erhebliche Vorteile mit sich bringen. Insbesondere in der Anforderungsanalyse und Qualitätsprüfung sind formale Methoden bereits ein etabliertes Werkzeug. Diese Bereiche der Softwareentwicklung betreffen die Qualität der Produktfunktionen. Ein weiteres Einsatzgebiet ist die Prozessqualität im Sinne einer Sammlung von Best Practices. Vorgehensmodelle wie ISO, Six-Sigma oder CMMI sind Leitfäden zur Optimierung von Geschäftsprozessen. Durch ihren Einsatz profitieren Organisationen, die komplexe Software entwickeln. In der Softwareentwicklung können die Ziele in den Einsatzgebieten Softwarequalität und Prozessqualität grundsätzlich unabhängig voneinander realisiert werden. Diese Arbeit soll jedoch die Möglichkeiten aufzeigen, wie mittels formaler Methoden das Vorgehen zur Verbesserung der Produktqualität und das Vorgehen zur Verbesserung der Prozessqualität integriert werden können. Der hierfür dargestellte Ansatz ist ein individuelles Framework zur Verbesserung der Produkt und Prozessqualität unter Einsatz von formalen Methoden.

Das beschriebene Framework basiert auf formalen Methoden und einem Prozessoptimierungsmodell. Das Framework unterscheidet in der Software-Spezifikation zwischen sichtbarem und internem Verhalten. Diese Unterscheidung erlaubt auf pragmatische Weise eine Unterteilung in Grob und Fein Spezifikation. Der formale Aufbau der Spezifikation folgt der erweiterten und angepassten Syntax und Semantik der formalen Sprache CSP-CASL. Das Vorgehen der Softwarespezifikationsverfeinerung wird um eine konstruktive Komponente erweitert. Bei dieser Herangehensweise werden bestehende Techniken der Softwarespezifikationsverfeinerung auf sichtbares Verhalten der Software angewendet. Das interne Verhalten der Software wird optimiert durch die Beschreibung des Software Designs. Das gesamte Softwarespezifikationsverfeinerung Framework basiert auf den Formalismen von CSP-CASL. Das Framework beinhaltet darüber hinaus ein Konzept zur funktionalen Erweiterung von Software. Die Erweiterung von Software als ein Prozess betrachtet, in dessen Verlauf neue Anforderungen an Funktionen oder Leistungsfähigkeit

in einer Software implementiert werden ohne die bestehenden Funktionen einzuschränken. Das Erweiterungs-Konzept wird formal ebenfalls beschrieben durch CSP-CASL.

Die erweiterte CSP-CASL Syntax wird dazu verwendet, sowohl positive als auch negative Testfälle zu generieren. Auf diese Weise wird sichergestellt, dass erwartetes und unerwünschtes Verhalten in den Testfällen enthalten ist. Ferner werden Testterminologien für die CSP-CASL Spezifikation beschrieben, welche Softwareverfeinerungen und Erweiterungen untersuchen. Anschließend wird die formale Definition von Softwareeigenschaften verwendet, um die Wiederverwendbarkeit von Testbestandteilen zu überprüfen. Diese Definitionen beschreiben die grundlegenden Eigenschaften im vorgeschlagenen Produkt und Prozessqualitäts-Framework. Ferner wird die Möglichkeit untersucht, die vorgeschlagenen Formalismus für die Entwicklung eines CMMI Prozesskonformen Frameworks zu verwenden. Dabei werden die Kernaspekte des CMMI Prozessmodells berücksichtigt. Das CMMI Compliance Bewertungssystem wurde entwickelt, um den Grad der Konformität der eingesetzten Softwareentwicklungsmethoden mit formalen Methoden zu bewerten. Ein generischer Algorithmus wird vorgeschlagen, um das Compliance Level der CMMI Prozessfelder und ihrer Komponenten zu ermitteln. Das Framework wird durch ein Tool unterstützt. Dieses Tool erlaubt es, die theoretischen Aspekte der vorgeschlagenen Theoreme praktisch zu unterstützen. Die Verwendbarkeit des vorgeschlagenen Frameworks wird an einem Anwendungsbeispiel aus der Medizintechnik gezeigt.

Im Rahmen dieser Arbeit wird das Verständnis der Anwendung von formalen Methoden auf das Organisatorische Prozessmodell CMMI erweitert. Das komplette Framework wird repräsentiert durch die formale Spezifikationssprache CSP-CASL sowie der Prozessmodell CMMI. Ähnliche Ergebnisse können auch mit anderen formalen Methoden und Prozessverbesserungsmodellen erzielt werden. Diese Forschungsarbeit dagegen bildet einen Startpunkt für eine Prozessmodellkonformität mit einen auf formalen Methoden basierenden Softwaresystems sowie deren Entwicklung und Wartung.

Chapter 1

Introduction

The ubiquitous presence of software systems requires incorporation of more and more functionalities which results in ever increasing complexities. The development and maintenance of such complex software systems pose new challenges for the software industries. Generally, a systematic approach is required to manage the complexity. This systematic approach is referred to as software engineering. More precisely, software engineering is a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of a software system [1]. Initially, the life span of software engineering was restricted to software system development and maintenance. However, the growing complexity and size of the software system requires software engineering at an organizational level. To manage the software system at the organizational level, various process improvement models have been investigated. Some of these process improvement models are CMMI [2], ISO 9000 [3], Six Sigma [4] etc.

The advantages of software engineering and process improvement models have been realized for a long time. But they have not become the de facto standard for the development of software systems. Generally, software engineering and process improvement models are neglected when other constraints (such as time, cost etc.) become bottle necks. The software industry faces a serious challenge to focus on software engineering and process improvement models for the software product development.

In this thesis, we explore the possibilities of necessary integration of formal methods, software engineering and a process improvement model. The complete approach is investigated as a framework of software product and process quality improvement. Product quality improvement is concerned with the wellness of capturing requirement, developing design, implementation and maintenance of software products. Software process quality is concerned with the monitoring of the software engineering related process at the product level as well as at the organization level. In this chapter, we first lay out the core issues associated with software system development which concern our thesis. Subsequently, this chapter presents an overview of the research as well as the scope and the objective of the thesis. The chapter ends by presenting thesis guidelines and by providing certain details of the subsequent chapters.

1.1 Software description

Inception of a software system starts by writing a precise description of software system behaviors. The description of the software system is referred to as a software specification or description. The complexity and variability of the software system bring challenges for the selection of a software specification notion. There are various notions of the software description but each of them possesses, along with good features, some additional constraints. In general, we can categorize the software specification notions as follows:

- Natural language based software description
- Graphical notation based software description
- Formal notation based software description

The presence of natural languages [5] has precedence for the description of the software systems over any other notions of software description. A natural language based software specification has the possibility of ambiguous interpretation, particularly when it is interpreted by many people. However, this is one of the most preferred techniques of software specification. Various research articles have shown that most errors in software development are due to the misinterpretation of software specifications [1]. A description might be interpreted differently if it is not written with precise semantics.

Graphical notation or formal notation based specification approaches have been developed with a concise set of rules. These rules are used to describe and interpret the requirements of the software system. There have been many approaches to graphical notation based software description widely known as graphical modeling languages [1] [6]. In this approach, Unified Modeling Language (UML) has made a significant contribution to the description and development of software systems. Today, UML is one of the preferred standard when it comes to specification, construction and documentation of a software intensive system. In 1997, UML has been proposed as a standard by Object Management Group (OMG) [7]. UML is known as a general purpose modeling language which has notions to specify, visualize, construct and document the artifacts of a software system under development. UML is not a development method by itself. However, it was designed to be compatible with object-oriented approaches of software development. The compatibility of UML with only object oriented approaches to software development restricts its applicability with other development methodologies. The graphical notation of UML semantics is imprecise which might lead to a subjective interpretation. This is one of the major problems which restricts the applicability of UML for automation of software development aspects.

The formal notation based software description is well-defined, complete, consistent, unambiguous and precise. Formal methods are based on mathematical notations which are key aspects for unambiguous syntax and semantics. Some formal methods are Z, VDM, CSP, CCS, etc. The formal methods have been considered to be very effective in the representation of a software description [8]. The increasing complexity of software systems demands a well-defined, complete, consistent, unambiguous and precise description. These demands are very well supported with formal method based software description. The formal method based description is referred to as formal specification. A formal specification allows describing a software system's behavior at various levels of abstraction. Since the formal specification has precise syntax and semantics it should allow to formulate a concrete relation between an abstract specification and its implementation. This relationship is investigated in various techniques such as refinement, rewrite rules and program transformations [9] [10] . In addition to this, formal methods can lead to an automatic verification and validation of formally developed software systems. Although a complete formal verification of a large complex system is impractical, formal methods are applied to various aspects or properties of large systems. More commonly, they are applied to the detailed specification, design, and verification of critical parts of large systems.

Formal methods have shown significant advantages, but their industrial uses are limited to certain domains. In this regard, formal methods are open for further research. In this thesis, we extend the understanding of formal methods in two directions. First, we present a pragmatic approach in the formalism of software systems. Second, we extend the understanding of formal methods to process improvement models. We start by exploring a distinct approach to software specification in order to tackle the pragmatic needs of software systems. We present an algebraic and process algebraic specification based formalism approach. In further sections, we give a general overview of the proposed approach and compare it with the existing research work.

1.2 Software evolution

In our consideration, the evolution of a software system is categorized in two types. In the first type of software evolution, the software system is stepwise developed by fixing a design decision with the consideration of its implementation. This type of software evolution is referred to as vertical software evolution. The second type of software evolution is a software system upgrade with new functionalities. The approach of adding new functionalities to an existing software system is referred to as horizontal evolution. In horizontal evolution, the existing software system functionalities are preserved, and new functionalities are added. Figure 1.1 presents a graphical view where software refinement and software enhancement are used as synonyms of vertical and horizontal evolutions, respectively.

Software evolution is not a new terminology for software industries. The importance of the software evolution has been realized since the proposal of software development life-cycle waterfall model [11]. In this software development life-cycle, the vertical evolution is considered to be part of the software development process. The horizontal evolution is taken into account after the software system deployment is finished. This means, the waterfall model was not flexible enough to handle the requirement changes in the middle of the software development stage. However, current demand of software systems has proved that software evolution

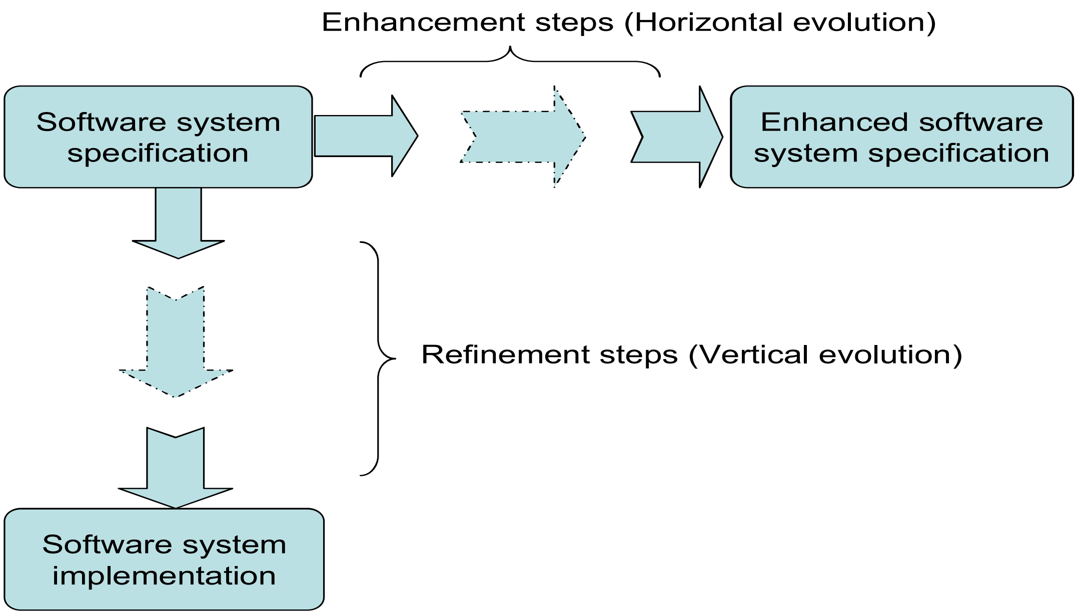

Enhancement steps (Horizontal evolution)
Software system specification
Enhanced software system specification
Refinement steps (Vertical evolution)
Software system implementation

FIGURE 1.1: Software system, Vertical and Horizontal evolution

is an integral part of any software system. At any stage of a software development life-cycle, software evolution support is required. There have been various methodologies to support such demands. Some of the well known methodologies are Agile software development, Model Driven Development etc.

The software enhancement has been studied as a product line [12] or as a component based development [13]. In a product line, software systems are developed from a common managed set of features from a specific domain of an industry. The strength of a product line is realized through time to market, cost, productivity, quality etc. Component based software development is initiated from a decomposition of software systems into functional and logical components. The decomposition is carried out with a consideration of reusability [13]. A product line and the component based development share the motto of reusability of software artifacts. There have been various research enhancements regarding the reusability of software artifacts such as [12][13]. But, the growing complexities and requirements of software systems demand new techniques to improve software products.

A systematic approach to vertical and horizontal evolutions is generally supported by graphical notation based specifications or by formal notation based specifications. The graphical notation based specification methodology is a semi-formal approach which restricts the possibilities of automation in vertical and horizontal evolution. Formal notation based software evolution has been investigated since the formal methods had been proposed. Various approaches to vertical software evolution such as software refinement [9], rewriting [14], program transformation [14] have been proposed. These approaches have made a significant contribution to the formalization of vertical software evolution. Until now most of the research has been carried out for vertical software evolution. Formal method based horizontal software evolution is still at the initial level of research. The existing scientific contributions to the software evolution are given in the further section of the thesis.

The challenge of software artifact reusability remains a major concern for software industries. Currently, software industries have invested a considerable amount of time and effort to develop software as well as software components, test cases, verification methods, validation methods and software correctness preserving techniques. Standard approach to artifact reusability is not a routine practice of software industries. Industries are always looking for a way to reuse existing software artifact from one application to another as well as within one application. In this thesis, we propose a formal specification, CSP-CASL based formalism for software evolution. Furthermore, software artifact reusability is explored with the proposed formalism of software evolution. The approach is elaborated with software evolution tool support. This tool allows one to understand the reusability of software artifacts during the software evolution.

1.3 Software quality

As software systems become more and more pervasive, there has been a growing concern about the software quality. The importance of the software quality has a different value for different types of software systems. In safety critical systems, there is no compromise on software quality, while in other applications, the software quality might be viewed differently. The definition of software quality is given as follows in IEEE Standard; *software quality is (1) the degree to which a system, component, or process meets specified requirements. (2) The degree to which a system, component, or process meets customer or user needs or expectations.* Software quality measures the wellness of design, implementation and maintenance properties of the software product. The quality of conformance is concerned with the monitoring of the software engineering process and with the methods used to ensure quality. The quality of design measures the validity of design and the requirements for the development of a software system. Generally, a software product life cycle is very long; unlike the life cycle of mechanical products where products are complete once they roll out from the assembly line. A software product lives, grows and evolves during its life cycle. In this regard, software systems require both product and process quality measurements which can be used to validate the software system's life cycle. In general, the software system quality management approach can be divided into the following categories:

- Software product quality
- Software process quality

This consideration has already been integrated into the most renowned software quality model known as the basis for all subsequent models. This model was presented by Jim Mcall [15]. This model has primarily aimed at software development and the software development process. This model brings users and developers together by focusing on a number of software quality factors that reflect both the view of user and the priorities of a developer. A graphical view of this model is presented in the Figure 1.2. The terms used are very generic in nature and are not elaborated in this context.

Currently, the importance of software quality is growing due to increasing penetration of the software systems. Software system functionality and its quality are the deciding parameters for existence of any software product. Inadequate software quality has been a major deciding factor for the failure of many software systems. The quality of a software system is evaluated via validation techniques. The validation process checks whether a system behaves as expected. To validate a software system, the desired behavior must be known. Two complementary validation techniques are testing and verification. The testing and the verification techniques are used to increase the level of confidence in the correct functioning of systems. Verification aims at proving properties about the system and testing is performed by exercising on the real implementation. Verification can give certainty about satisfaction of a required property, but this certainty only applies to the model of the system: any verification is only as good as the validity of the system model [16]. Testing is based on observing only a small subset of all possible instances of system behavior and is usually incomplete. Testing can show

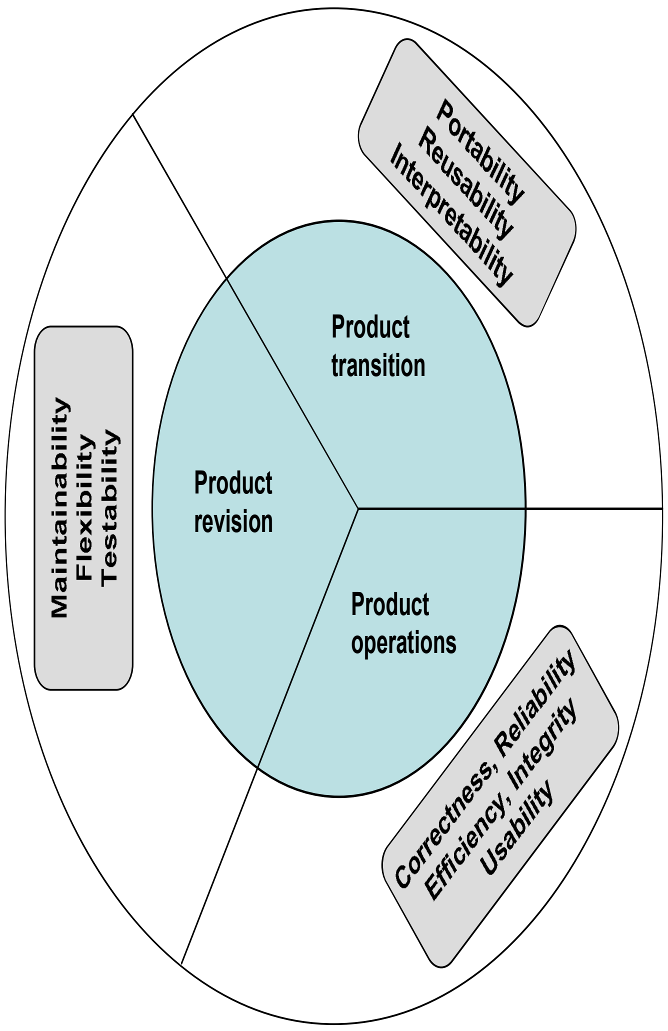


Figure 1.2: Prominent software quality model

the presence of errors but not their absence [17]. The main prerequisite for testing and verification is the description of the expected behavior of the software system.

Formal methods have shown significant benefits in the testing and verification techniques of the software systems development [18] [19] [20]. However, formal methods are not completely understood for process quality. In this thesis, we consider process quality as a compliance of an organizational process model [2], CMMI. Our aim is to develop a formal method based product and process quality framework as shown in figure 1.3. The framework proposes to start formal method based software system development. Furthermore, software development, product quality and process quality should be governed with formal method based techniques. In brief, product development, product quality and process quality are managed with a single base of formal methods.

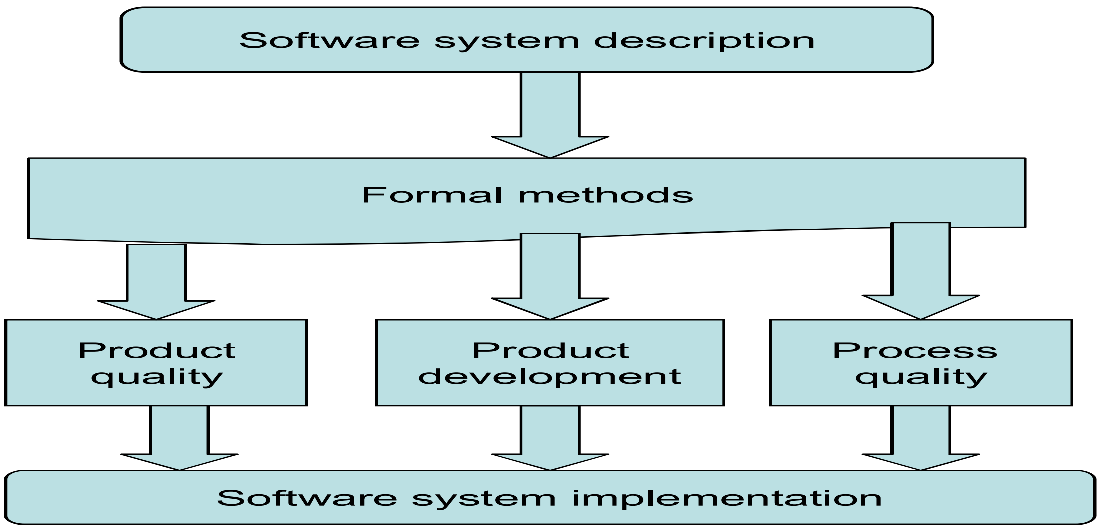


Figure 1.3: Formal methods, Product and Process Quality

The proposed process and product quality framework is based on the formalism of software specification, software refinement and software enhancement. By giving formal definition to these concepts, we develop a product and process quality achievement framework for the development and maintenance of software systems. As software process quality, we analyze the compliance of the CMMI [2] process improvement model. CMMI (Capability Maturity Model Integration) is a well-established process improvement model which has proved its benefits in hundreds of companies and in thousands of projects. This model, however, does not enforce or suggest a specific approach to the product development processes. However, for the compliance of a process model systematic approach is required. Our proposal to the systematic approach is formal methods based system development and maintenance. Figure 1.3 shows a conceptual presence of formal methods into the product and process quality of a software system. This conceptual representation shows that formal methods are introduced as soon as requirements are drafted. They are present throughout the life cycle of a software system. In this diagram, the life cycle of a software system is represented with product quality, process quality and product development. However, it is not clear in which way the two aspects, i.e. a process quality and a product quality, influence each other. In this research, we discuss the pros and cons of formal method based development into the compliance of CMMI practices within an organization. Our integrated approach of formal method based development in a CMMI environment can be used as a guide to achieve a sustainable process and product quality.

1.4 About this thesis

The main aim of this thesis is to develop a product and a process quality improvement framework for the development and maintenance of software systems. Particularly, this framework will be based on the uses of formal methods for the software development. First, the research proposes a distinct approach to software specification. This specification approach is further evaluated into the software systems life cycle. The software system life-cycle is concerned with software life-cycle (refinement) as well as software system family development (enhancement). The proposed framework is developed with an algebraic and process algebraic specification language CSP-CASL [21]. In the further subsections we will give an overview of the proposed product and process quality improvement framework.

1.4.1 Scope and objectives

Software product quality is determined by processes used to evolve and to develop it [2]. However, in many situations, development processes are given lower priority due to unavoidable constraints such as time, budget, complexity etc. In our research we aim to develop a framework for product and process quality of a software system development. Here, we briefly explain the scope and objectives of our research. Specifically, we elaborate the boundaries and the accomplishment plans of our research.

Foundation of this research is built on the formal methods. Formal methods have contributed significantly to the development and quality improvements of software systems. However, they have not been significantly explored for the possibilities to improve the software system process quality. Particularly, in this research we explore the possibilities of formal methods into software product and process quality improvement. We extend understanding of a software specification as an integration of observable and internal specifications. Formulation of this approach is proposed with a syntactic extension to the existing specification language, CSP-CASL .

This specification formalism is based on an integrated language of process algebra and algebraic specification language. The syntax of CSP-CASL specification language is also extended for the specification of failures of software systems.

Further, the proposed specification formalism is explored for the syntactic and semantic definition of software refinement and enhancement. The extended specification formalism is evaluated for a distinct approach of test case generation. The possibilities of positive and negative test case generation are explored within our proposed formalism. A test evaluation algorithm is proposed for a CSP-CASL specification formalism. The generated test case properties are further explored for software refinement and enhancement. Our approach of test evaluation is restricted to the specification itself. The test evaluation properties are explored on software specification, its refinement and enhancement. The test generation, evaluation, and reusability of test cases in the specification refinement and enhancement contribute significantly to the product quality framework.

As a process quality improvement framework, the compliance of a process improvement model is evaluated. Particularly, the properties of formal methods are investigated for process model compliance. As a process model we have selected CMMI which has been quite successful for the development of software systems. To evaluate the compliance with formal methods based software development, a compliance grading scheme is proposed. The grading scheme is further applied into the proposed compliance algorithm to evaluate the extend of CMMI process area compliance. Our proposed grading schemes assist into standard CMMI compliance evaluation methodology SCAPMI (Standard CMMI Appraisal Method for Process Improvement) A/B/C [22] to decide the level of CMMI compliance to an organization. These grading schemes are not an alternative to the CMMI SCAMPI methods, which are standard approach for an evaluation of CMMI compliance in an organization.

The proposed process and product quality improvement framework is supported with a developed tool. This tool helps to keep track on many aspects of software development which forms the basis for the quality improvement framework. As a proof of concept, an industrial case study is developed and evaluated within our proposed framework. However, the complete implementation of CMMI process requires many projects which is out of the scope of our current research. At this stage, we can only elucidate the idea of combining process and product quality improvement with our proposed quality framework.

1.4.2 Thesis guideline

This thesis is structured as follows. Chapter 2 introduces the terminologies adapted from various literature such as publications and books. The adapted terminologies are briefly described within the scope of this thesis. This chapter starts with general details about formal methods and their categories. Then syntax and semantics of the algebraic specification language CASL and the process algebra CSP are described within the scope of required details. Further, some process models are described which are used in industries for the development of software systems with a consistent and controlled approach. Formal methods and process models are subsequently co-related to product quality factors. The remaining part of chapter 2 presents an overview of related research work.

Chapters 3, 4 and 5 represent the main contribution of this thesis. Parts of these chapters have already been published in scientific articles and they all are listed into the our publication list. Chapter 3 introduces the foundational part of this thesis by giving details of our proposed terminologies of software specification, refinement and enhancement. These terminologies are defined based on the specification formalism of process algebraic and algebraic specification language. We introduce a distinct approach to software specification, which is based on observable and internal behavior of a software system. On the basis of this specification, the formalism of software refinement and software enhancement is defined. The given definitions are supported with a case study which describes the applicability of the given definitions.

Chapter four presents a product quality framework. Particularly, in this chapter, testing methodologies for the CSP-CASL based specification formalism are investigated. Our previously defined formalisms of software specification, software refinement and software enhancement are used to propose testing terminologies in the vertical and horizontal software development paradigm. Test generation and evaluation are supported by a tool, which is developed to support a formal method based product quality framework. Furthermore, this tool also provides a basis for process model compliance as will be discussed in the upcoming chapter.

Chapter five presents a process model compliance framework which is one of the very important achievement of this research work. Particularly, the CMMI process model is elaborated and some basic definitions are given to measure the compliance level with the formal method based product development. The complete details of CMMI process compliance with formal method are presented. The compliance algorithm is presented which helps to evaluate compliance level of CMMI process areas, specific goals and specific practices.

Chapter six presents the results and the prospective of our research. After this chapter, the thesis is concluded with references and other obligatory sections.

1.4.3 Case study: MED overview

To demonstrate our approach, we worked with an industrial partner for design, development and maintenance of a Medical Embedded Device (MED). The MED is a monitoring and control unit of a mechanical heart support system developed by our industrial partner. An overview of the heart supporting device is shown in Figure 1.4. Particularly, this diagram represents a broad view of this device which includes MED, as a controlling and monitoring unit. The functionalities of this device are controlled by a control unit which is referred to as *Controller* and is shown at the lower part of the Figure 1.4. This part of the MED is designed to monitor and control the human heart depending on the health of the patient. The scope of our project was restricted with the configuration and monitoring of MED by different mechanisms. Being a safety critical system, the importance of data should be precisely considered before any data related activities are carried out. Data integrity is a major concern, particularly for the transmission of patient's data. To achieve the above requirements of this safety critical system, we propose to use a formal method to design and develop this Medical Embedded Device.

We demonstrate our research activities with a small part of the MED, which is particularly concerned with the communication interface of the MED with other devices. The function of the communication interface is divided into two parts: configuration and monitoring. Being a safety critical system, access and manipulation of patient's data must be handled with appropriate care. With this consideration, the first version, *(Basic MED)* provided a configuration and monitoring facility with a serial interface of the computer. The *Basic MED* is developed in such a way that patient's data is sent with appropriate encryption to the respective departments (such as hospitals, monitoring centers, doctors) for the analysis, monitoring and controlling of a patient's health.

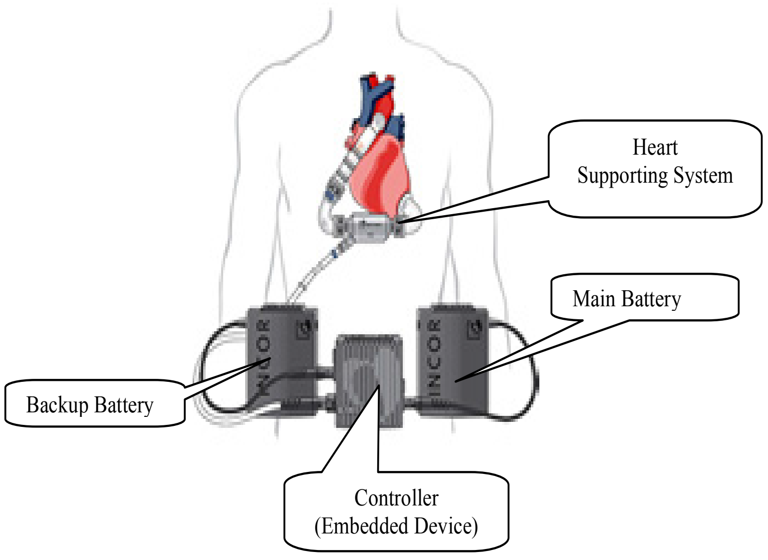


Figure 1.4: Overview of heart supporting device

Basic MED: To ensure data integrity, a customized encryption algorithm must be developed for the exchange of patient's data. Any communication with the MED should start with an acknowledgment of a reliable connection with connected device. The developed communication protocol should assure that patient's data is always transmitted in an encrypted format. In this device, communication with any computer should only be possible through a serial interface.

Enhanced MED: However, technological developments and patient's needs have been incorporated with the evolution of MEDs. In the first advanced version of the *Enhanced MED*, various types of connections were proposed. Particularly, due to the advancement in web technologies, the *Enhanced MED* was incorporated with additional connection possibilities of ethernet and dial up connections. A brief overview of the MED is shown in the Figure 1.5.

In the *Basic MED*, communication was only possible via a serial interface. This means, the patient had to be in hospital for the controlling and monitoring of her or his health. This restriction was waved off, in the *Enhanced MED*, by incorporation of different connection mechanisms. Figure 1.5 shows an architectural overview of MED. This self-explanatory figure shows details of MED communication possibilities. In further chapters we shall elaborate the development of this MED using our proposed framework.

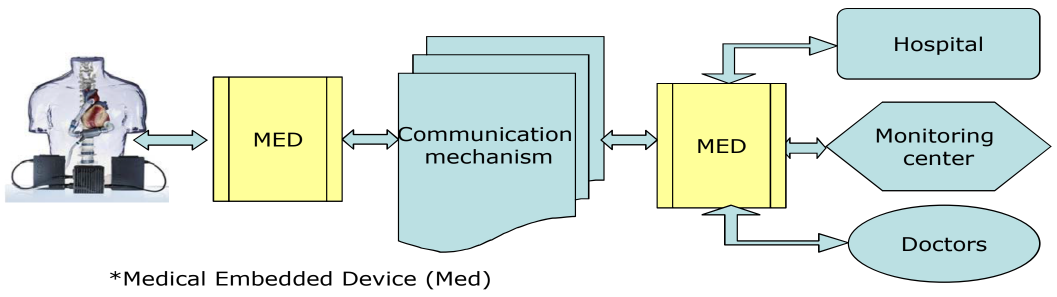


FIGURE 1.5: Medical Embedded Device (MED) architecture

Chapter 2

Preliminaries and related work

Research work is always based on existing foundational researches. However, the citation of all related work is nearly impossible. This chapter includes references to the existing researches which are related to the context of this thesis. In the first part of this chapter; the contributions of formal methods to the software engineering domain is elaborated. Particularly, the syntax and semantics of algebraic and process algebraic specification languages are briefly explained.

Further, some known process improvement models are briefly discussed. This part of the chapter is concluded by conceptualizing a relation between formal method and the process improvement model. A pictorial view of this relationship is presented to give a general understanding of this concept. For the next part of this chapter, we describe related research work and their state of art. Especially, software refinement, software enhancement, software product and process quality related researches are elaborated within the confinement of this research. Additional details of related research work are cited whenever they are used within thesis.

2.1 Formal methods

Formal methods have been a focus of software engineering research for many years and they have established advantages in various stages of the software development life cycle. Formal methods are based on mathematical techniques of specifying and verifying software systems. They are better known for specifying complexity of software systems in a well-defined, complete, consistent, precise and unambiguous manner. In particular, formal methods have been applied at various stages of software development life cycle. Generally, they are well known for specification, development, verification, semi-automatic and automatic proofs. The uses of formal methods can be categorized into three levels: [8][23]

- **Level 0 - Formal specification**
- **Level 1 - Formal development and formal verification**
- **Level 2 - Theorem proving**

A formal specification of a software system is expressed in a language made of three components: rules for determining the grammatical well-formedness of sentences (syntax), rules for interpreting sentences in a precise, meaningful way within the domain considered (semantics) and rules for inferring useful information from the specification (proofs) [1]. In specific terms, a formal specification is a mathematical description of the software system that can be used to develop an implementation. This is fundamental (level 0) use of the formal method. At level 1, the formal method is used for the formal verification techniques to demonstrate that system design is correct with respect to the given formal specification. A formal development of a software system starts from initial formal specification (level 0) and all future design steps until the implementation are validated with the formal methods based techniques (level 1 and 2).

The formal methods can be classified according to their approach of software system specifications. This research related approaches to software specifications are algebraic specification [7], model-based specification and process algebraic specification. In an algebraic specification approach, first operations of specifying system are identified; further their behaviors are captured by describing relationships among these operations. The description of software system behaviors are referred to as axioms. Examples of algebraic specification languages are Larch, Obj, CASL.

Another approach of specification is model-based specification, which is often considered to be a more concise specification approach. A model-based specification provides a model of system's state as a system's state model. The state model is constructed using mathematical entities such as sets and functions. System's operations are defined in terms of how they modify the state model: pre-conditions define valid input and the start state for an operation and post-condition defines output and state of system after operation execution. Some widely used notions for developing model based specifications[24] are VDM[25], Z[8]. The formal method of this group is further divided for the specification of sequential and concurrent systems. Process algebra is a formal description technique for complex software systems, especially those involving communicating, concurrently executing components. Well known examples of process algebra are CSP [8], CCS [23], ACP[26] , and LOTOS [27]. This thesis evolves around the features of algebraic specification language CASL and process algebra CSP. This combined language has been found most suitable for elaboration of our research concept. Particularly, this combined language gives opportunity to investigate our research in the broad spectrum of formal methods from data type development to the concurrent process evaluations.

2.1.1 CASL, Common Algebraic Specification Language

The specification language CASL is developed under Common Framework Initiative (CoFI) for algebraic specification and development of software. CASL is consolidation of previous works on the design of algebraic specification languages [28][29][30]. This specification language is well suited for writing formal specification of functional requirements and modular software system design.

The CASL provides basic specifications, structured specifications, architectural specification and library based specifications to describe software systems into various layers. The CASL basic specification allows declaration of sorts, operations, predicates and axioms as the first order formula. Some of the CASL keywords used for writing basic specification are **sort/sorts** (data type), **op/ops** (operation), **pred/preds** (predicate), **var/vars**(variable). These symbols are self-explanatory, however they are explained in the later phases of the thesis whenever they are referenced.

A CASL structured specification is a combination of basic specifications into a larger specification in a hierarchical and modular fashion. It mainly allows translation (keyword **with**), reduction (keyword **hide**), union (keyword **and**) and extension (keyword **then**) of specifications. An architectural specification allows systems to be developed as reusable components by describing the way modules/components are to be combined. A library based specification allows easy distribution and reusability of components. A detailed description of algebraic specification language CASL and its semantics can be found in *CASL User Manual* [28] and *CASL Reference Manual* [29].

Like any algebraic specification language, a CASL specification SP is denoted as a tuple of (Signature Σ, Axioms E). The CASL many sorted signature Σ is tuple of (S, TF, PF, P), where:

- S is set of sorts
- $TF_{w,s}$ is total function, with profile over sequence of arguments $w \in S^*(finite)$ and result sort $s \in S$
- $PF_{w,s}$ is partial function, with profile over sequence of arguments $w \in S^*$ and result sort $s \in S$. A function without argument is considered as a constant symbol
- P_w is a predicate symbol, with profile over sequence of arguments $w \in S^*$

The semantics of CASL basic and structural specification are class of models over the signature which satisfies all the specified axioms. The class of models of specification is a subset of $Mod[\Sigma]$ such that it satisfies all the axioms E. For a given many-sorted signature Σ, a many-sorted model is interpreted by assigning respective values to each symbols. For a signature Σ; a many sorted model $M \in Mod[\Sigma]$ has the following interpretation:

- non empty carrier set s^M to each $s \in S$
- a total function $(f)^M : w^M \rightarrow s^M$ for each function symbol $f \in TF_{w,s}$
- a partial function $(f)^M : w^M \rightarrow s^M$ for each function symbol $f \in PF_{w,s}$
- a predicate $(p)^M$ for each predicate symbol $p \in P_w$

A many sorted Σ homomorphism $h : M \rightarrow N$ is a family of functions $h = (h_s : M_s \rightarrow N_s)_{s \in S}$ with the property such that all $f \in TF_{w,s} \cup PF_{w,s}$ and $(a_1, .., a_n) \in s^M$ with $f^M_{w,s}(a_1, .., a_n)$ defined, we have

$h_s((f_{w,s})^M(a_1,..,a_n)) = (f_{w,s})^N(h_{s1}(a_1),..,h_{s1}(a_n)))$

and for all $P \in P_w$, and $(a_1,..,a_n) \in s^M$,

$(a_1,..,a_n) \in (P_w)^M$ implies $(h_{s1}(a_1),..,h_{s1}(a_n)) \in (P_w)^N$.

Let, $\sigma : \Sigma \to \Sigma'$ be a many sorted signature morphism, M' be a Σ' model. Then the reduct $M'_{|\sigma} := M$ if M' is Σ model with

- $s^M := (\sigma^s(s))^{M'}$ for all $s \in S$
- $(f)^M := (\sigma^F(f))^{M'}$ for all $f \in TF_{w,s} \cup PF_{w,s}$
- $(p)^M := (\sigma^p(p))^{M'}$ for all $p \in P_w$

$M \models \varphi$ holds for many sorted Σ model and a many sorted first order formula, iff $v \vdash \varphi$ for all variable valuations v into M [29].

The specification SP is referred to as consistent if models of SP are non-empty. The specification language, CASL is developed for the specification of structural properties of system; however dynamic properties are frequently required for the complete specification of any system. This limitation of CASL is overcome by the integration of CASL with another specification languages such as CCS and CSP [31]. The integrated specification language well suited for a specification of static and dynamic behaviors of specifying systems. In the further subsections, we give a brief overview of process algebra CSP and its integration with CASL.

2.1.2 CSP, Communicating Sequential Process

CSP is action based formalism for describing and analyzing reactive systems. It provides a set of mathematical symbols to model complexities of reactive systems with clarity and preciseness. Action based formalism allows direct communication among components of the system. CSP defines this direct communication as events of the system.

In CSP, an object $(e \to P)$, is described with an event e and process P, which states that the object first engages in an event e and then behaves like process P. The operator $\to$ is referred to as prefixing operator which leads to a concept of recursion. For example, the process $P = (e \to P)$ will be continuously willing to participate in the event e. Two primitive CSP processes are *STOP* and *SKIP*, where *STOP* is a process which never engages in any event and *SKIP* is a process which does nothing but terminates successfully. The process $P = e \rightarrow SKIP$, offers the communication e and then behaves like *SKIP*, hence it terminates successfully.

Table 2.1: CSP syntax

$P :=$	STOP	no event is accepted
	SKIP	successful termination
	e $\rightarrow$ P	prefixing
	?x:X $\rightarrow$ P	prefix choice
	P [ ] P	external choice
	P |~| P	nondeterministic choice
	P ||| P	interleaving
	P |$\{X\}$| P	interface parallel
	P $\backslash$ X	hiding
	P ; P	sequential composition
	if φ then P else P	boolean conditional

CSP has a rich syntax for describing processes; some of the syntax which are used in this thesis is given in the Table 2.1. It involves element $e \in A$ as communication, subset $X \subseteq A$ as synchronization set used in parallel operator or for hiding certain communications, and unspecified formulae φ in its conditional statement. Let X be a set of communication, then $?x : X \rightarrow P(x)$ is a process which will communicate any value $x \in X$ and then behave like $P(x)$. This choice operator allows the choice among the values to be communicated. We can also write the same with the channeled version as $c?x \rightarrow P(x)$, to send a value over channel we write as $c!x \rightarrow P(x)$.

The process $P;Q$ is a process which behaves like P, once P terminates, then it behaves like process Q. For an example: $P = e \rightarrow SKIP$ and $Q = f \rightarrow SKIP$, then the process $R = P;Q$ is equivalent to $e \rightarrow f \rightarrow SKIP$. CSP supports two types of choice operators; external and internal choices. The external choice operator, $P[]Q$ offers the environment the choice of first communication of P and Q, and then behaves accordingly. For example: ExChoice= $(e \rightarrow SKIP)$ [] $(f \rightarrow STOP)$, if the environment offers e, then process *ExChoice* will communicate e and then successfully terminates. Otherwise, if environment offers f, the *ExChoice* will communicate f and then deadlock. In the internal choice operator, choice is made in nondeterministic way. The interleaving form of parallel combination is supported by CSP where processes don't communicate with each other. Nondeterminism is an important feature for writing an abstract specification of communicating systems. In nondeterminism a process can choose amongst several alternatives for its further elaboration. Hiding of events is supported for abstraction; this may also lead to nondeterminism. Further details are given in respective examples whenever they are used. The syntax shown in Table 2.1 seems self-elaborative, however additional details of these syntax can be found in various research articles and books such as [32] [33]. CSP offers three distinct approaches to semantics. These semantics are known as:

- algebraic semantics
- operational semantics
- denotational semantics

Algebraic semantics of CSP allow definition of the most abstract specification by a set of algebraic laws. The operational semantics interpret a CSP program as transition diagrams. The denotational semantics map a language into abstract model. Further denotational semantics of CSP is represented based on the behaviors such as traces, failures and divergences [32]. Our research is based on the denotational semantics, further we give a brief overview of this semantics.

CSP, trace model denotes a process according to its traces, which are set of sequence of communications in which a process is willing to engage. Let $A^{*\checkmark} = A^* \cup \{s^\frown \langle \checkmark \rangle \, | s \in A^*\}$ be alphabet of communications, where $\checkmark \notin A$ represents the event of successful termination. In the trace model each process is identified by a set $T \subseteq A^{*\checkmark}$. This set must satisfy two healthiness conditions. One, T is nonempty, it always contains empty set $\langle\rangle$ and two, T is prefix closed, $e^\frown f \in T$ then $e \in T$. For a given process P, the traces of P is denoted by $traces(P)$. In Table 2.2, we report the semantics of a trace model T.

The trace model of CSP is not capable of distinguishing traces of internal choice and external choice. To overcome such issues of trace model, stable failure model, failure divergence model has been proposed. A failure of process is a pair (s,X), that describes sets of communications X which a process can fail to accept after execution of trace s. The set X is referred as refusal set. The process will not perform any events of set X, no matter how long it is offered. More details of these can be found in a very well-known book on CSP by Roscoe [32]. In the next

TABLE 2.2: CSP semantics clauses for trace model

traces(STOP)	$\{\langle\rangle\}$
traces(SKIP)	$\{\langle\rangle, \langle\checkmark\rangle\}$
traces(e → P)	$\{\langle\rangle\} \cup \{\langle e\rangle^\frown s \mid s \in traces(P)\}$
traces(?x:X → P)	$\{\langle\rangle\} \cup \{\langle e\rangle^\frown s \mid s \in traces(P[e/x]), e \in X\}$
traces(P [ ] Q)	$traces(P) \cup traces(Q)$
traces(P |~| Q)	$traces(P) \cup traces(Q)$
traces(P || Q)	$traces(P) \cap traces(Q)$
traces(P \ X)	$\{s \backslash X \mid s \in traces(P)\}$
traces(P ; Q)	$(traces(P) \cap A^*) \cup \{s^\frown t \mid s^\frown \langle\checkmark\rangle\ s \in traces(P), t \in traces(Q)\}$
traces(if φ then P else Q)	traces(P); if φ true, else traces(Q);

subsection we would like to give an overview of a process algebraic specification language CSP-CASL.

2.1.3 CSP-CASL

CSP-CASL is a language which combines the description of structural and behavioral properties of software system. In a formal specification of a software system, the processes are specified in the CSP and communications between these processes are the data type values which are described in CASL. The syntax of CSP-CASL is an integration of CASL and CSP syntax . This integrated syntax is limited to the CASL basic and structural specification constructs. Syntactically, a CSP-CASL specification **Sp** consists of a data part **D**, which is a CASL specification, an (optional) channel part **Ch** to declare channels, which are typed according to the data specification, and a process part **P** written in CSP, where CASL terms are used as communications. Thus, a generic syntax of a CSP-CASL specification is:

$$\text{ccSpec } \mathbf{Sp} = \text{data } \mathbf{D} \text{ channel } \mathbf{Ch} \text{ process } \mathbf{P} \text{ end}$$

In the semantics of CSP-CASL , the loose semantic nature of CASL induces a family of process denotation. The complete semantics of CSP-CASL is defined in three steps. In the first step each channel is encoded in CASL. In the second step CASL data types are evaluated, where families of processes are generated according to the data model of CASL. In the last step the evaluation according to the CSP takes place. The definition of language CSP-CASL supports all possible

CSP semantics. However, in our thesis we consider denotational semantics of CSP. More details of this formalism are described in further chapters of this thesis.

2.2 Process improvement model

Modern technology helps to solve computational demands of the software system development. However, modern technology should be supported with various other activities to complete the software system development lifecycle. The needs of processes were realized from the early 70s, for the development of software systems with consistent qualities (such as reliability, efficiency, evolvability, ease of use, etc.). It was realized that these qualities could only be injected in the software systems by following a disciplined flow of activities. Afterwards, the software process was recognized by researchers as a specific subject. Later, this field deserved a special attention and dedicated scientific investigation to understand its foundations, develop useful models, identify methods, provide tool supports etc.

The software process is a set of activities for specifying, designing, implementing and testing software systems. At the initial step, software process focus was more on the software engineering aspects. Software engineering focus was to produce quality software products through quality processes. Some of the well accepted software process models were proposed and further they evolved as de facto standard for the development of software products. Such software process models are waterfall model, prototyping, evolutionary development, formal systems development, reuse-based development etc. The proposed models were quite successful for the development of manageable software products. Further, software products became part of all industrial domains which made software products unmanageable only with initially proposed software process models.

Software process models were further enhanced with a consideration of organizational activities of software product development. The process models which have shown significant benefits are ISO, Six-sigma and CMM/CMMI. Initially ISO 9000 and Six-Sigma were applied to mechanical and electrical domains. Further, their advantages were recognized by software industries. Initially, CMM/CMMI was used in software industries and then was adopted by other industries. In the further subsections, brief overviews of these models are described.

2.2.1 ISO

ISO (International Organization for Standardization) ensures that managed products and services are safe, reliable and of good quality. In an ISO certified organization errors are minimized, which subsequently increases productivity. ISO has proposed various types of process models for different industrial domains. A well-known quality management system ISO 9001:2000 has a very significant proposal to condense and harmonize the goal of common applications. The core aspects of this quality management system are:

- Understand the requirements
- Establish processes to meet those requirements
- Provide resources to run the processes
- Monitor, control, and measure the processes
- Improve continuously based on the results

These are considered best quality programs for customer satisfaction, or rather to meet customer expectations. Software organization related quality management program, ISO/IEC 15504 is formal reference to SPICE (Software Process Improvement and Capability dEtermination) model which was developed as an international initiative in 1995. ISO 15504 is a specially designed framework for assessment of processes. This contains a reference model of process dimension and capability dimension. This standard defines an approach to conformity of the reference model. ISO 15504 process dimension defines customer-supplier, engineering, supporting, management, organization level processes. Capability level defines the scales such as Optimizing process, Predictable process, Established process, Managed process, Performed process and Incomplete process. SPICE currently has a narrower focus on the development aspect of software. Most of the features of ISO 15504 are included in CMMI process compliance model.

2.2.2 Six-Sigma

Six-Sigma is a business performance measurement strategy process, initially proposed by Motorola and further enhanced and refined by GE [4]. Six-Sigma seeks to identify and remove the causes of defects and errors in product development. Six-Sigma has recently been adopted by the software industry which is looking for better software products within a controlled environment. Six-Sigma is different from ISO 9001 and CMMI in the sense that, it focuses on the measurement of existing processes with a view to make them more efficient and effective. Six-Sigma assumes that processes are in place and they are formally or informally applied through the process. At its core, Six-Sigma is a way to measure processes and then modify them to reduce the number of defects found in the produced products. In this aspect, Six-Sigma is different from ISO 9000 and CMMI family but it brings a lot of mathematical measurement into the practice. Statistically, the measurement of Six-Sigma means that your system will turn out only 3.4 defects per million opportunities for defects. The main idea behind Six-Sigma is to manage process improvement quantitatively. Six-Sigma acts as an evaluation side to a process improvement program which makes it fit together with other process models such as ISO 9001, CMMI etc.

Six-Sigma uses two basic methodologies to problem solving. The first is referred to as DMAIC. DMAIC is used to improve existing processes in an organization. The other methodology is DFSS (Design for Six-Sigma). DFSS is used to design a new process and introduce it into an organization in a way that it supports Six-Sigma management techniques. There are five basic steps in the methodology known as DMAIC:

- Define
- Measure
- Analyze
- Improve
- Control

It is a process improvement methodology that employs incremental process improvement using Six-Sigma techniques. DFSS methodology also has five steps: define, measure, analyze, design, verify. These terms are known with their names and more details can be found [4]. The CMMI process model compliance can assist into seamless implementation of Six-Sigma process.

2.2.3 CMM/CMMI

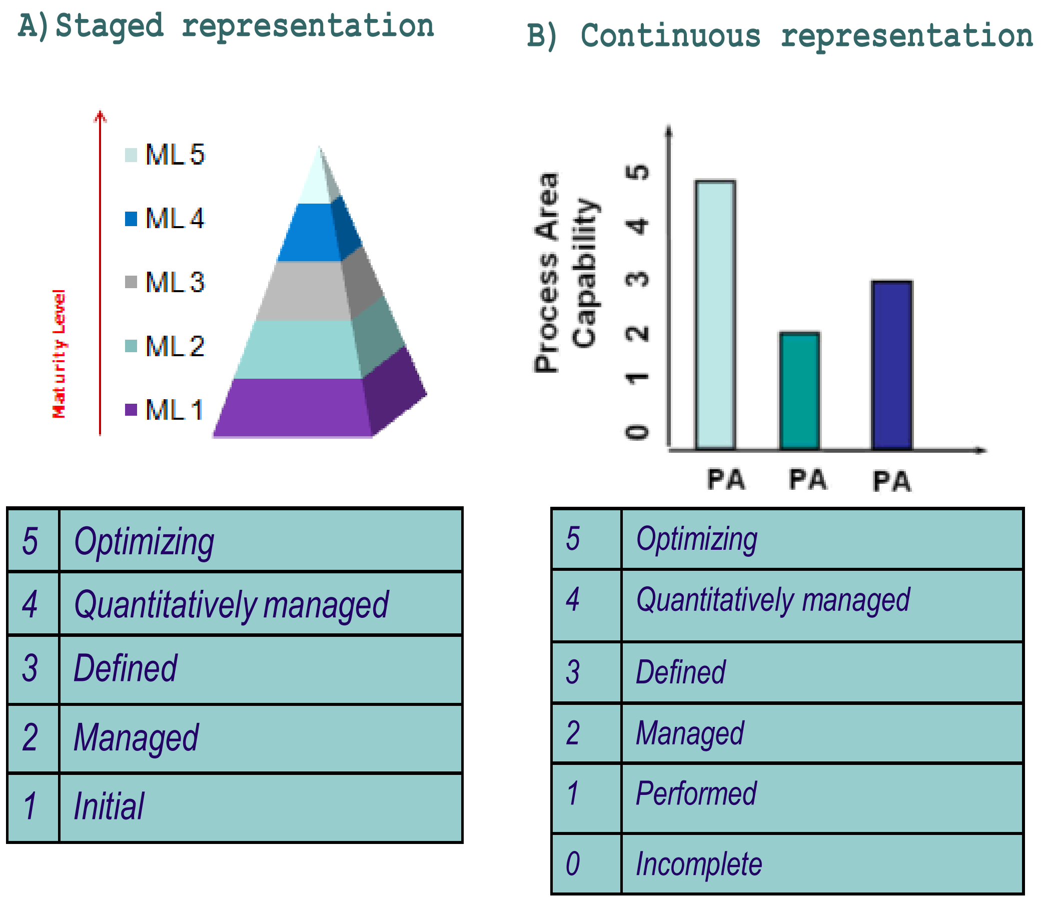
Capability Maturity Model (CMM) was better known as a software development process improvement model. When CMM is applied into a software development organization, it helps to understand and improve the capability and maturity of software development processes. The process model CMM, was first described into the book *Managing the Software Process* [34] by Watts Humphrey. This concept was fully elaborated in his book *Quality is Free* in 1979. However, the active development of the model started in 1986 by US Department of Defense Software Engineering Institute (SEI). From 1987 till now, there have been many versions of this model which have undoubtedly benefited thousands of projects and hundreds of organizations. Currently, CMMI (Capability Maturity Model Integration (CMMI)) Version 1.2 [35] released in 2006 is a process improvement framework for software engineering and organizational development. Now days, a single organization usually does not develop all the components of a product and services. In such case, organizations must be able to manage and control this complex development and management process. These types of issues are integrated in the proposal of CMMI which has broadened the applicability of CMM.

The CMMI defined best practices, that can be used in a project, or in a department or in an entire organization to improve the chances of business success. CMMI is designed to be used in three different areas of interest:

- product and service development (CMMI for Development)
- service establishment, management, and delivery (CMMI for Services)
- product and service acquisition (CMMI for Acquisition)

CMMI representation allows an organization to pursue its process improvement objectives with two different approaches. In CMMI terminology they are referred as *Staged representation* and *Continuous representation* as show into figure 2.1 A. The staged representation uses predefined sets of *process areas* to define an improvement path for an organization. This approach of *process areas* compliance is referred to as Maturity level. A maturity level is a well-defined evolutionary plateau towards achieving improved organizational processes [2]. The figure 2.1 A, shows the set of maturity levels which an organization can achieve successively.

On the other hand, in CMMI *Continuous representation* an organization selects *process areas* according to their expertise in business functions. Then they improve selected *process area* up to certain capability level. The CMMI capability levels are shown into the Figure 2.1 B.



5	Optimizing
4	Quantitatively managed
3	Defined
2	Managed
1	Initial

5	Optimizing
4	Quantitatively managed
3	Defined
2	Managed
1	Performed
0	Incomplete

FIGURE 2.1: CMMI representations

The benefits of CMMI process implementation have been demonstrated both quantitatively and qualitatively. The organizations that have adopted its recommendations and consciously applied them within their projects have seen measurable performance improvements. Project planning, estimates and projections have become more accurate. Work paths have become more established. Efficiencies have increased. Defect rates and rework have dropped.

Particularly, CMMI for development consists of processes that address development and maintenance activities and applies to products and services. These processes cover the product's life cycle from conception through delivery and maintenance. The emphasis of our research is related to the CMMI for the development. More details on this process model are given in further chapters of the thesis.

2.3 Product and process quality

The previous sections have been dedicated to an introduction of required preliminaries. On one hand, we described the formal methods and on the other hand, we described process improvement models. Both of them are well known for the improvement of product quality. Generally, product and process qualities are considered with different prospects. However, their goal is to develop a better software product. A process is a set of activities which have some coherence and whose objective is generally agreed within an organization. Generally, the collections of processes are known as a process model. All process models have their own set of objectives. Such process models have shown significant benefits for the development of software systems with systematic and controlled process. On the other hand the formal methods are better known for the improvement of product quality. The product quality is related to the functional aspect of software systems. Subsequent part of this thesis, explores relationships among software quality factors. Figure 2.2 presents the factors which are important to develop a quality software system.

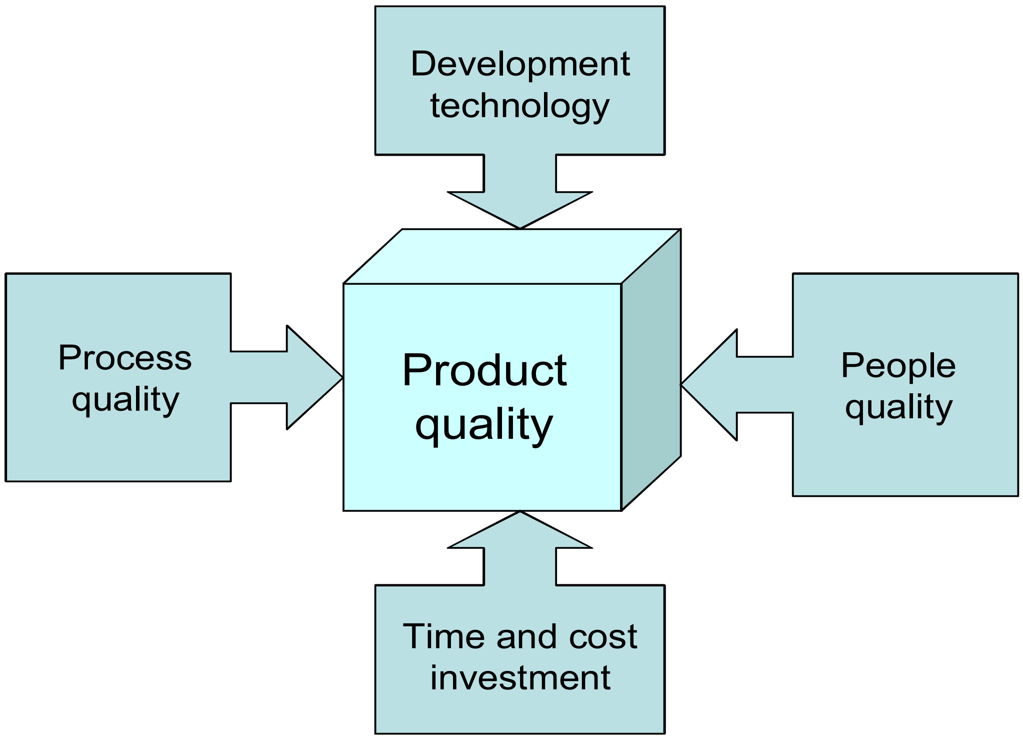


Figure 2.2: Product development and quality factors

The presented quality factors explain that software product development is not only dependent on the development technology. In addition to the technology, a good process is usually required to produce a good product. Not only process and product but people, cost and time also play important roles for the development of a quality product [36]. People can be motivated with good process and technology. Cost and time can be reduced with proper use of process and technology. This means that the fundamental aspect for software system quality is process quality and product quality. Generally, the product quality and process quality are considered two separate aspects for a software product development and maintenance. Some tools are used for the software development life cycle and some other tools are used for the process quality, particularly for the compliance of used process models. In our thesis, we investigate the possible integration of product and process quality with formal method based software development.

2.4 Related work

The software quality improvement researches are well-known for decades. There have been significant achievements which are necessary to be taken into consideration before the proposal of any new research. We consider various aspects of existing state of art in the software product and process quality improvements. Particularly, formal method based software development has been elaborated in various research articles. Our research has extensively used algebraic specification techniques which have been explored by various well-known research experts such as [37] [38] [39] [40]. These articles are the basis of our research work. Specific aspects of formal methods based research techniques are briefly discussed in the further subsections. In a similar fashion, process models have been real paradigm for the improvement of software products. We have considered CMMI [34] as a process improvement model to elaborate our research. In this section, we explore state of art of various other methodologies which have significantly contributed to product and process improvements. Particularly, we give a brief overview of product and process quality within the limitation of our research boundaries. Additional related research articles are cited whenever they are used throughout the thesis.

2.4.1 Specification refinement

Inception of a software system starts with an abstract specification, which is progressively transformed into an implementation. The steps involved from an abstract specification to an implementation are considered as steps of refinement [41][42]. Each refinement step is based on some correctness preserving techniques. The first article on refinement notion was written by Edsger W. Dijkstra, titled as *Constructive approach to the program correctness*. A simple example to the constructive approach of a program correctness can be given as follows:

Example 1. For a program S and postcondition x, $wp.S.x$ is the weakest condition that must hold prior to the execution of S to ensure that S terminates in a state satisfying x.

The spectrum of Dijkstra's research was diversified by Niklaus Wirth [43] as a concept of *stepwise refinement*. Further, a logical foundation of refinement was contributed by Rod Burstall [44], John Darlington [45] and C.A.R Hoare [46]. Dijkstra's weakest precondition calculus was also extended by Ralf-Johan Back [47] with the consideration of preserving total correctness instead of partial correctness between the program statements. Back and Von Wright elaborated the refinement calculus as a logical framework for reasoning about the program. The objective of refinement calculus is related to improving the program while preserving its correctness. The refinement calculus is formalized within higher order logic which allows to prove program correctness and program refinement. This notion of refinement is given by a pre-condition/post-condition pair. For every pre-condition P and post-condition Q, if S validates post-condition Q assuming pre-condition P then T refining S validates also post-condition Q assuming pre-condition P. This definition is further extended to data refinement and for the parallel and reactive action systems by the author [48].

Carroll Morgan [49] and Joseph Morris extended the refinement calculus framework with different specification statements. Morgan published a book about program refinement, in this book he has elaborated the concepts of refinement with various case studies. Further, there have been various researches for extending the applicability of refinement calculus to various domains e.g. parallel programs [41], reactive programs [50], object oriented programs [48], model based program [10], algebraic specification based program [30]. Most of the refinement calculus related researches use an imperative programming language based on Dijkstra's guarded commands language, specification constructs and weakest precondition semantics. Refinement calculus provides a foundation for the development of software tools in the provable journey of correctness preserving transformations [51]. Existing work of refinement is based on the principal's of top-down design; where the program development begins with writing a specification and subsequently developing executable program by series of correctness preserving transformations.

A description of software system requires both structural and behavioral properties. Similarly, an implementation of such description requires refinement of structural and behavioral properties as well. The refinement of behavioral properties is reduction in expressiveness and nondeterminism; refinement of structural properties is concerned with the replacement of data types with simpler and more efficient implementable type. There have been various refinement framework for data and process refinement such as [52][10][43][51]. In this thesis, we elaborate a concept of structural and behavioral property refinement with the extended CSP-CASL specification language. State of art for these specification language refinement properties is initially described. Subsequently a pragmatic approach to CSP-CASL refinement is presented. The presented refinement is an extension of existing refinement approach with a consideration of observable and internal behaviors of specified software system. Here, we briefly describe the various well known techniques of specification refinement which are directly or indirectly related to our research.

2.4.2 Refinement Calculus

Refinement calculus is a theory of program development. It includes a wide spectrum of programming languages to express specifications and executable programs. Refinement calculus consists of a refinement relation capturing the notion of correct program development and collection of laws expressing possible program development steps. The refinement relation is defined, and refinement laws are proved in terms of underlying semantics of the programming language. The refinement laws then allow the program to be constructed and verified at the programming language level. Various notion of refinement calculus have been important topics of research, some well-known researches are the works of authors Ralf Back and Joakim von Write, Carrol Morgan, Ken Robinson, Joe Merris etc. The programming language used in these research articles is a non-deterministic imperative language based on Dijkstra's guarded command language. The refinement relation is defined as the weakest precondition semantics. More details on refinement calculus can be found in books such as [47][51]. Our research is more concerned with the process algebra and algebraic specification based specification languages and their refinement formalisms.

2.4.3 Action refinement

In the context of process algebra, a well-known refinement approach is often referred to as action refinement. The concept of action refinement was introduced by Ursula Goltz and R.J. van Glabbeek [53] inspired by the design of concurrent systems. The design of concurrent system is based on the actions which may occur in a system. An action is any activity which is considered as a conceptual entity on a chosen level of abstraction. This allows the representation of systems in a hierarchical way, changing the level of abstraction by interpreting actions on a higher level by more complicated processes on a lower level. This change of the level of abstraction is referred to as refinement of action.

Action refinement is defined as the refinement of action as sequential execution of several subactions, activities happening independently in parallel or an action can be refined as a set of alternatives. This type of refinement is proposed in such a way that behavior of refined system can be inferred compositionally from the behavior of the original system. The initial aim was to propose a suitable refinement operator independent of any specific model for the description of concurrent system. Some theoretical constraints have bounded this refinement approach suitable to Petri nets [54].

Other process algebra CCS do not include the sequential composition operator. Thus in order to support action refinement, action-prefixing is usually replaced with sequential composition. CSP has been well known for its different semantics and refinement theories. CSP has been quite successful for the development of a commercial tool [23], to analyze the refinement relation between CSP specifications.

2.4.4 Data refinement

Niklaus Wirth [43] proposed that process and data should be refined in parallel to construct a program with a sequence of refinement steps. The research of data refinement was further elaborated by C. A. R. Hoare as a powerful method of simplifying the proofs of program correctness [46]. His paper presents an automatic method of achieving the transition between an abstract and a concrete program and proposes a method of proving its correctness. The proof method is proposed by giving relationship between abstract and concrete representation as a function which maps the concrete variable into the abstract object. The proof of program correctness was more formally described in algebraic terms by Milner [55]. Gardiner and Morgan in 1993 proposed a practical definition for the refinement as follows: a given pair of programs called concrete and abstract, the concrete program refines the abstract program correctly whenever the use of the concrete program does not lead to an observation which is not also an observation of the abstract program. Let's explain this with an example as given in the book [55].

Example 2. Let $S1$ and $S2$ denote statements, which are not involving variables s and l. The following two programs represent a refinement relation:

begin	**begin**
var s : finset of N ; s := null;	var l : finset of N; l:= null;
S1;	S1;
s:= $S \cup x$;	l:= append(l,x);
S2;	S2;
y:=a member of s;	y=first(l);
end	**end**

In this example, refinement steps are achieved by replacement of variable and operations. The variable s is refined by another variable l, similarly, the operations are refined by *append* and $first$. In this refinement approach, variables are categorized as normal variable and data representation variable according to the abstract level. This is a way to achieve a concrete program which is a refinement of abstract program. Further, downward and upward simulation are proposed as the sound techniques for proving data refinement [52].

Based on the collection of various research articles, the following general methods were proposed in the book of data refinement:

- One or more concrete variables are introduced to store the representation of one or more abstract variables.

- A general invariant called the representation invariant is introduced, which describes the relationship between the abstract and concrete variables.
- Each assignment to an abstract variable (or more generally, each assignment that affects the representation invariant) is augmented with assignments to the concrete variables that reestablish the representation invariant (or achieve it, in case of an initialization).
- Each expression that contains an abstract variable but occurs outside of an assignment to an abstract variable is replaced with an expression that does not contain abstract variables but is guaranteed by the representation invariant to have the same value.

Data refinement laws are basic requirements for refinement of an algebraic specification based software description. Algebraic specification is known for writing abstract specification of software systems. Abstraction in specification allows to hide implementation details from the users. The abstract specification is further refined to an implementation level by following some rules of provable journey. In the next subsection, we present the existing refinement approaches of algebraic specification which are the basis for the proposal of our research.

2.4.5 Algebraic specification refinement

The theory of algebraic specification started with the pioneer work of Goguen [44], Guttag [56] and Ziles [45]. The two important aspects of algebraic specification are theories of formal specification and notion of refinement. There have been various notions of algebraic specification and refinement such as [30] [29] [57] [9]. Algebraic specification transformation into programs has been extensively presented into the project PROSPECTRA (PROgram development by SPECification and TRAnsformation) [58]. A significant contribution by Donald Sannela is a starting point for the further development of algebraic specification and refinement notions [37]. His paper subsumes most of the algebraic specification based research activities and extends the concept of loose semantics as well as initial semantics of algebraic specification.

In algebraic specification, programs are modeled as many-sorted algebras consisting of collection of sets of data values (S) together with operations (Ω) overs the sets of data types. The properties of specification are given with axioms which are some type of first order logic with equality. The data types and operations of a specification represent the signature(Σ) of specifying systems. The class of (Σ)-algebra is denoted by $Alg(\Sigma)$ and the class of (Σ)-homomorphism from A to B is denoted by $Alg(A) \rightarrow Alg(B)$. A homomorphism between two specifications allows to establish relation between the specifications. Algebraic specification is particularly known for the specification of Abstract Data Type (ADT), which consists of a family of data types sharing abstract properties usually given by some sort. For any abstract data type, there can exist many specifications characterizing to that abstract specification. There are various notions of algebraic specification refinement; below we present refinement notions of algebraic specification language CASL:

- **A simple refinement:** SP' is refinement of SP (represented as $SP \rightsquigarrow SP'$) iff SP' incorporates the requirements that any realization of SP' is correct realization of SP. This relation is given as follows with an assumption of $Sig(SP) = Sig(SP')$ [29]:
  $SP \rightsquigarrow SP'$ iff $Mod(SP') \subseteq Mod(SP)$

- **Constructor based refinement:** Simple refinement is sufficient to prove requirement relation when the specifications are sufficiently rich. However, during stepwise refinement, successive specification is built with more and more design decisions. Some parts become fixed and the remaining parts of specification do not change until a concrete program is obtained. The concept of constructor is proposed to tackle the finished part of specification separately and then to proceed with unresolved parts. It is defined as follows:
  Suppose SP and SP' are specifications and k is constructor such that
  $k : Alg(Sig(SP')) \rightarrow Alg(Sig(SP))$, then:
  $SP \rightsquigarrow (k)SP'$ iff $k(Mod(SP')) \subseteq Mod(SP)$

- **Decomposition based refinement:** Modeling of program with a separate subtask is achieved by constructor implementation with multi-agent constructor as presented in paper [59]. Formally it is shown as follows:
  $SP \rightsquigarrow (k)(SP_1, SP_2.., SP_n)$ iff $k(Mod(SP_1, SP_2.., SP_n)) \subseteq Mod(SP)$

2.4.6 Software enhancement

Business organizations are always required to upgrade existing software systems to support their business function changes. The expectation is always to integrate the changes into their software systems within minimum expenditure and time frames. This could only be possible with reuse of existing software systems and their artifacts. However, there is a major challenge to reuse the running software systems artifacts and upgrade them to support the business dynamism. There have been many research activities to describe the approach of software evolution in terms of software enrichment, product line based development etc. Nevertheless, software evolution is not commonly known and practiced for software artifact reusability.

M M Lehmen wrote in his paper [60] as a first law of the software evolution as follows:

- If the same software is operating for a long time with same functionalities, it progressively becomes less satisfactory

Further, he proposed various rules of software evolution; even today we find his laws of evolution are not fully addressed with theoretical details and tool supports. It is very commonly observed that software systems evolve to tackle the requirements of day to day business needs. Generally, software evolution is not only addition of more functionality, but also very often architectural changes, re-engineering, and modifying existing functionality is considered as a part of software evolution. Specially, we investigate the concepts of specification enhancement with formal specification based software development. Our consideration of software enhancement is addition of new features to the existing software specification by preserving its existing features.

Formal methods have been known for describing specification in a structured manner. Generally, structuring of specification allows to investigate software enhancement. Particularly, this concept has been elaborated in algebraic specification language [28]. Algebraic specification proposes three types of operations for building specification in a structured manner. These structuring methodologies are taken as a basis for an investigation of specification enhancement. In our thesis, we closely describe a formal approach of adding features and functionalities to the existing software system. The formal understanding of the software evolution is described with an integrated process algebraic and algebraic specification language. At the initial step we formally define a syntax and semantics of software enhancement which is further investigated as a concept of software artifacts traceability and reusability.

- **Enrichments:** This is an approach to enhance a specification by adding more elements(e.g. sorts, operations) to an existing specification. A given specification SP is enriched by $(\Sigma 1, E1)$. Then the new signature of complete specification is $(sig(SP) \cup \Sigma 1)$ and a set of equations are based on the signature $(sig(SP) \cup \Sigma 1)$. Often, enrichments are required to preserve the initial specification properties. This is satisfied as follows:
  $Model(Enriched\ SP)|_{SP} = Model(SP)$
- **Union:** An incremental specification building is supported by union of two smaller specification in an algebraic specification formalism. Suppose $(\Sigma 1, E1)$ and $(\Sigma 2, E2)$ are two small independent specification (no common specification) then the resultant syntax of specification is $(\Sigma 1 \cup \Sigma 2, E1 \cup E2)$ which has semantics similar to simple algebraic specification. Union of two smaller specifications can also have some sub-specification as common to both the specifications. The common specification leads to a complexity which is handled by renaming in algebraic specification. The complete detail is not related to our thesis however, this is described in the research article by Fernando Orejas [61].
- **Derive:** Sometimes expressiveness of existing specification has to be enhanced which gives better understanding to the existing specification. However, this over-specification can be avoided by hiding extra sorts and operations of specification, by using (hide keyword) to hide these auxiliary elements. For a given specification $SP = (\Sigma, E)$ the meaning of *derive* $\Sigma 1$ *from SP* is specification $(\Sigma 1, E1)$, such that $E1$ is the set of all $\Sigma 1$ equations which are logical consequences of SP. Model of derived specification is represented as $(\Sigma 1, Model(SP)_{|\Sigma 1})$.

These are the very common features for any algebraic specification languages. We have considered this type of specification manipulation for an investigation of a new definition of software enhancement. Our proposed definition is based on the structural and behavioral specification languages; which provides more power to describe software systems in precise manner.

2.4.7 Software product quality

Software product quality is concerned with correct implementation of the expected software system behaviors. Software system behaviors are captured as user requirement. If a software system behavior is not properly known, then, there is very little possibility to validate the system's expected behavior. A validation is the process of checking the system's behavior with respect to the given system requirements. A description of desired behavior is called specification; it describes what the system must do and not how to do. A system that is supposed to implement desired behavior is called implementation, e.g. embedded system, software etc. Here, based on this understanding, we define validation as a process of checking whether implementation complies to its specification.

Generally, a validation process is supported by two complimentary techniques such as verification and testing. Verification aims at proving properties about system from a mathematical model of the system. On the other hand, testing is performed by executing the implementation or its executable model. Verification limits its existence to the model which might be different from real implementation. But verification has been found very useful where testing cannot be done on the implementation. In this thesis, we want to be more close to the implementation or its executable model. Our concern is related to the investigation of testing properties. Current state of research has different importance to testing and verification. People from the realm of verification very often consider testing as inferior, because it can only detect some errors, but it cannot prove correctness; on the other hand, people from the realm of testing consider verification as impracticable and not applicable to realistically-sized systems.

Testing has its presence at different levels of the software development life cycle such as unit testing, component testing and system testing [62] [63] [48]. Each level of testing contributes to the validation of the final software system. For each type of testing, different aspect of tests can be performed such as stress, performance, conformance and acceptance tests. In this thesis, we concentrate on conformance testing based on the formal approach to the software system specification. Generally, test cases are derived from the formal specification and the derived test cases are used to test implementation or the executable formal specification. There have been various research activities of these types of testing. Here, we briefly cite state of art of testing advances and relations with our proposed approach.

Formal methods have shown significant benefits for an automatic test generation [16] [19] [20] and test evaluation [64]. The formalization of software system behavior is the starting point for the formal specification based testing. A formal specification of software system behaviors provides various means to generate and evaluate the test cases. Some of the well-known formal specification languages for these contributions are Estelle, LOTOS, or SDL. However, to test a software system, access to the implementation is required. Broadly, this can be achieved with three types of access to the implementation for the testing purpose. Based on the access type, testing can be divided into three categories; white-box testing, gray-box testing and black-box testing. In white-box testing internal structure of an implementation is fully known. In black-box testing it is assumed that the implementation can only be accessed through its interface with the environment, and no knowledge of the internal structure of the implementation is known. Gray-box testing lies in between black-box testing and white-box testing. In gray-box testing it is assumed that only part of the internal structure of an implementation is known. More details on the test generation and test evaluation is delineated in subsequent part of this thesis, particularly, in the chapter on product quality framework.

2.4.8 Software process quality

A software process is a method of developing software systems. Applicability of software process has shown significant benefits for the development software system, particularly in the improvement of product quality with various quality factors. McCall [15] in 1977 presented a categorization of quality factors into 11 aspects e.g. efficiency, integrity, reliability, usability, accuracy, maintainability, testability, flexibility, re-usability, transferability etc. The elaboration of these quality factors can be found in the paper [15]. However, these terms are well known for the society of software quality. The systematic approach is required to achieve quality factors for a software product development. Software process quality is an approach of process conformance for the production of required quality products. Our consideration of process quality is confined to the compliance of process model with formal method based software development approach. There have been significant contributions on the formalization of development process by Basin and Krieg-Brückner. In their book, the techniques for the transition from requirement specification to verified design are elaborated.

Formal methods are well known for refinement process in the software development, where they provide a provable journey from abstract specification to an implementation of the abstract specification [65] [44]. Software enrichment has been also established by many researchers where a formalization has been proposed [44]. UML based software process quality has been the focus of software industry which has achieved significant maturity in practice as well as in theory. Our approach of formal method based process quality is a distinct approach where software process compliance starts with investigation of formal methods based software development.

2.5 Summary

In this chapter, a brief overview of required and related techniques is given. As the required techniques, formal methods and process improvement models are concisely elaborated. There have been various types of formal methods for a specification, analysis, verification and validation of software systems. We have selected process algebraic specification language CSP-CASL for our research. This specification language is an integration of process algebra CSP, and algebraic specification language CASL. This specification language has been considered appropriate for the specification of dynamic and static behaviors of software systems.

As the process improvement models, first we briefly elaborated the three well-known improvement models such as ISO, Six-Sigma and CMMI. Further, these process improvement models have been co-related with each other, to analyze the best of their approaches. In our research, we have selected the CMMI process improvement model, this process model subsumes required improvement guidelines for any software system development within an organizational environment.

As the related research work, we have briefly described the techniques of formal methods based refinement and enhancement. The software refinement is well known in the research community. We have briefly explained various techniques of refinement which are directly, or indirectly related to this thesis. In addition to this, presence of formal methods with regard to product and process quality of software system is briefly sketched out. At the end of this chapter, the concepts of software product and process quality is briefly elaborated.

Chapter 3

Software evolution methodology

Today software systems are everywhere, from medical applications, automobiles, telecommunications to large enterprise management systems. Software is no longer a few hundred lines of code, but several hundred million lines of code to control complicated systems. To manage the life cycle of such software systems a systematic approach is required. A software system life cycle is categorized into two steps. In the first step, the software system evolves through software development phases and in the second step, the software system evolves with the addition of new features. To tackle these situations, notions of vertical and horizontal evolution have been proposed. Vertical evolution and horizontal evolution are also known as software refinement and software enhancement.

Vertical evolution of a software system is the process of developing a software system from a given abstract specification. Software system development from an abstract specification to an implementation is achieved with the steps of refinement. At the each refinement step, abstract specification is fixed with design decisions by preserving its correctness from the previous step.

Horizontal evolution of a software system is the process of adding new features to the existing software system. This type of evolution is referred to as software enhancement where new features are added by preserving its existing features. These types of software evolution have been in practice. However, a precise understanding of software evolution for software system development and maintenance is still an active area of research. Particularly, it is still not clear what happens to the existing software artifacts when a software product evolves vertically or horizontally.

In the initial sections of this chapter, a pragmatic approach to software specification is described. In the proposed methodology, we distinguish observable and internal behaviors in the description of software systems. First, these concepts are formally described and then based on the given definitions; a software evolution methodology is presented. Particularly, software refinement and software enhancement are defined based on the specification of observable and internal behaviors of software system. Subsequently, these definitions of software refinement and software enhancement are explored in the evolution of the proposed case study of the medical embedded device.

3.1 Software specification

Inception of software system starts by gathering customer requirements in a clear, complete, consistent, traceable, modifiable manner. The customer requirements are generally broken in one or more functional requirements, where exact software system functionalities are documented. The growing number of functions of a software system increases the complexity in the description of software functionality. A function is described as a set of inputs, the behavior, and outputs [66]. Functional requirements are supported by non-functional requirements also known as quality requirements. Functional requirements are distinguishing characteristics of any software system. The non-functional requirements are also a subset of software features. In IEEE 829, the term *feature* is defined as a distinguishing characteristic of a software item (e.g., performance, portability, or functionality).

Generally, at the initial stage, software requirements are abstractly described with a broad overview of system. In further steps, such an abstract specification must be deterministic to be accepted by computers. The abstract specification should evolve in such a way that it preserves the abstract properties and adds precise implementation descriptions.

To give a formal approach to abstract and detailed specification, an observable and internal behavior based software specification technique is presented. This is very much related to a practical approach of developing a software system, where first an abstract idea of software system is described and in the subsequent stages implementation details are added. This approach of software specification allows to track changes of specification during evolution of software system. In the further subsections, we describe the fundamentals of our methodology and its application for the formalism of software system evolution.

3.1.1 Observable and internal behavior

A software system is a consolidation of observable and internal behaviors. The description of observable behaviors is confined to the functional requirements of a software system. The observable behaviors are further elaborated with design decisions for the purpose of implementation. These design decisions are considered as internal behaviors, and they are included into observable behavior of specifying system. In the initial stage of software specification, internal behaviors are omitted to provide implementation freedom, such specification are considered as abstract specifications.

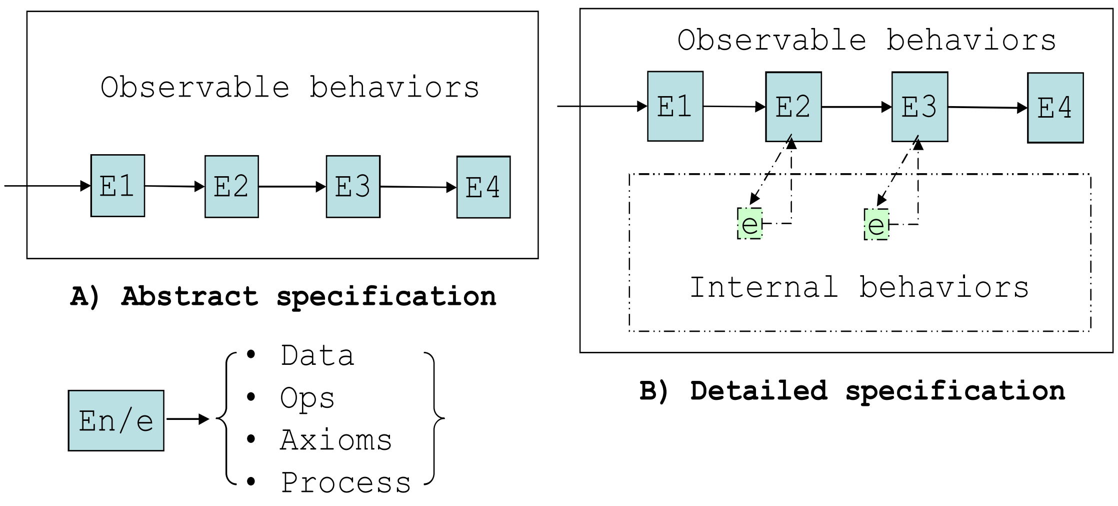


FIGURE 3.1: Observable and internal behaviors of a system

In this approach, at the initial stages, a software system is abstractly specified as observable behaviors and further design decisions are added as internal behaviors. In general, at the initial steps of software specification, only observable behaviors are specified as a precise and complete description of software system. Further, the internal behaviors are included to elaborate the design decisions. In the Figure 3.1, we give a rudimentary example to present a concise overview of observable and internal behavior of a software specification. At the initial step only observable behaviors ($E1, E2,\ E3, E4$) are described as functional requirements of the specifying system. At the design level, observable behaviors are more deterministically represented by addition of internal behaviors (e). It is assumed, that the functional requirements are completely captured at the initial stage and abstractly represented as observable behaviors. Internal behaviors are only required to elaborate implementation details for the observable behaviors.

The distinction between observable and internal behaviors of a software specification allows a pragmatic approach for the refinement and enhancement of software system. This approach allows to add internal behaviors at the steps of refinement which is practically required for the implementation of an abstractly specified software system by preserving specification correctness and fixing the design decisions. The approach of observable and internal behavior based specification can be elaborated with any specification language. This concept is not constrained to any specific approach of specification. However in this research, we elaborate this concept with an algebraic/process algebraic specification language CSP-CASL .

3.2 Formal software specification

A formal specification based technique is presented to describe the observable and internal behaviors of a software system. The selection of specification language is based on our experience with specification language which is suitable for the specification of any kind of software system. The selected specification language is particularly advantageous for an analysis of data type development in the process modeling of a software system. The foundation of proposed specification language is based on the specification language CSP-CASL . The CSP-CASL (Communicating Sequential Processes-Common Algebraic Specification Language) is a specification language; developed with an integration of process algebra (CSP) and algebraic specification language (CASL). The complete syntax and semantics of this specification language has been described in the paper [21]. In our specification technique; we extend the syntax of CSP-CASL to support the software specification as observable and internal behaviors. The extended syntax of CSP-CASL is formulated as follows:

$$\textbf{ccSpec}\ \mathit{spName} = \textbf{Data}\ D_{obs}\ D_{int}\ \ [\textbf{Channel}\ Ch_{obs}\ Ch_{int}]\ \textbf{Process}\ P_{obs}\ P_{int}\ \textbf{End}$$

In the above syntax, keywords are represented in bold face. Here *spName* represents the name of a CSP-CASL specification. The complete syntax is consolidation of data specification, channel specification and process specification. The data part specification is an integration of observable D_{obs} and internal D_{int} data specification which is specified by algebraic specification language CASL. To distinguish the observable and internal part of specification, we propose to append an underscore (_) sign in the declaration of internal specification syntax. Ch_{obs} and Ch_{int} represent observable and internal communication channels in the system specification which are optional parts of specification. The last part of specification is the process part where dynamic behaviors are described as observable P_{obs} and internal processes P_{int}. The extended syntax of CSP-CASL is also presented in the Figure 3.2. The CASL keywords *And* and *Then* are borrowed to represent the integration of observable and internal data part specifications. The keyword *And* allows to integrate an independent internal data part into the existing data part of observable behaviors. The observable $dataSpec_{obs}$ and internal $dataSpec_{int}$ data parts are specified with the CASL syntax as given in the manual [67]. In this research, we are restricting our approach to CASL basic and structured specification. This thesis does not provide complete CASL syntax, however all the required keywords are explained whenever they are required.

Data
    $\text{dataSpec}_{\text{obs}}$ And/Then $\text{dataSpec}_{\text{int}}$

Channel
    $[\text{channelSpec}_{\text{obs}}]$
    $[\text{channelSpec}_{\text{int}}]$

Process
    $\text{processSpec}_{\text{obs}}$; $\text{processSpec}_{\text{int}}$

End

FIGURE 3.2: CSP-CASL specification syntax

$processSpec_{obs}$ and $processSpec_{int}$ represent CSP processes which describe the dynamic behaviors of specifying software system. A CSP process is described by one or more events followed by a process. An event may be atomic or associated with data. The complete syntax and semantics of CSP process are described in [1][9]. In this approach to specification, we propose a syntactic extension into CSP syntax to describe failures into specification. In the extended CSP syntax a process $P := P_{cs}[F_e]$, where F_e is set of actions P_{cs} can refuse. P_{cs} is CSP syntax as shown in the Table 2.1. This syntactical extension is further elaborated in the formalism of our case study. An observable process $processSpec_{obs}$ is described with CSP syntax where $dataSpec_{obs}$ is used as data for communication. Similarly an internal process $processSpec_{int}$ is described with CSP syntax where the communication between processes is the value of data types specified by $dataSpec_{obs}$ or $dataSpec_{Int}$. A keyword $';'$ is concatenation syntax for the CSP processes and is borrowed from basic CSP syntax which allows to concatenate internal and observable processes. In the description of a software system, we confine our approach to denotational semantics. Within this semantics, we consider only traces and failures denotation of process which are sufficient to describe safety and liveness properties. Traces of a given CSP process P; $traces(P)$ are set of sequence of events and $failures(P/t)$ is set of events which process P can refuse to perform after performing a trace t. For a CSP-CASL specification formalism, CSP-CASL traces are categorised as **Complete trace** and **Incomplete trace**. A complete trace is sequence of events where last event of the trace successfully terminates according to definition of [32], or last event of the trace is in the set of defined failures. An incomplete trace is the trace of the expected behavior where last event of the trace does not terminate successfully [32]. Furthermore, the syntax and semantics of this definition remains the same as described in the paper [68]. A Figure 3.2 represents CSP-CASL based syntactic formalism by presenting a distinction between data, channel and process parts.

For simplicity of the specification formalism, we have not considered channel part; in this case a CSP-CASL syntax can be represented as a tuple $(D_{obs}, D_{int}, P_{obs}, P_{int})$. The data part (D_{obs}, D_{int}) is described with the CASL syntax and process part (P_{obs}, P_{int}) is described with CSP syntax with an integration of data part. At the abstract level of specification, this syntax reduces to (D_{obs}, P_{obs}) which has similar syntax as proposed by [21]. The data part is described by algebraic specification language CASL, where a specification is a tuple of (Σ, E), such that $\Sigma{=}(S, TF, PF, P)$ is many sorted signature and E is set of axioms over signature Σ. The complete details of syntax and semantics can be found in the CASL reference manual [29]. In this research, totality and partiality of a function is not distinguished, the main focus is to elaborate our concept with less complexity in specification. In this consideration CASL signature is written as $\Sigma{=}(S, F, Pr)$ where S is set of sorts, F is total or partial operations and Pr is set of predicates. A CASL specification of (D_{obs}, D_{int}) can be represented as tuple of $(\Sigma_{obs \cup int}, E_{obs \cup int})$, where $\Sigma_{obs \cup int}$ is tuple of $(S_{obs}, S_{int}, F_{obs}, F_{int}, Pr_{obs}, Pr_{int})$.

Signature: A many sorted signature $\Sigma_{obs\,\cup\,int}$=$(S_{obs}, S_{int}, F_{obs}, F_{int}, Pr_{obs}, Pr_{int})$ consists of

- S_{obs} is a set of observable sorts
- S_{int} is a set of internal sorts
- $F_{obs(w,s)}$ is a function, with function profile over sequence of arguments $w \in S^*_{obs}(finite)$ and result sort $s \in S_{obs}$
- $F_{int(w,s)}$ is a function, with function profile over sequence of arguments $w \in S^*_{int} \cup S^*_{obs}$ and result sort $s \in S_{int} \cup S_{obs}$
- $Pr_{obs(w)}$ a predicate symbol, with profile over sequence of arguments $w \in S^*_{obs}$
- $Pr_{int(w)}$ a predicate symbol, with profile over sequence of arguments $w \in S^*_{obs} \cup S^*_{int}$

A signature morphism defines a mapping from the sort, operation, predicate names in one signature to another signature. The signature morphism allows to establish a relation between two signatures where values of signature symbols are preserved. Given two signatures $\Sigma_{obs\,\cup\,int}$=$(S_{obs}, S_{int}, F_{obs}, F_{int}, Pr_{obs}, Pr_{int})$ and $\Sigma'_{obs\,\cup\,int}$=$(S'_{obs}, S'_{int}, F'_{obs}, F'_{int}, Pr'_{obs}, Pr'_{int})$, a many sorted signature morphism $\sigma : \Sigma_{obs\,\cup\,int} \rightarrow \Sigma'_{obs\,\cup\,int}$ is represented as follows:

- a map $\sigma^S_{obs} : S_{obs} \rightarrow S'_{obs}$
- a map $\sigma^S_{int} : S_{int} \rightarrow S'_{int}$
- a map $\sigma^F_{obs(w,s)} : F_{obs(w,s)} \rightarrow F'_{obs(\sigma^{S*}_{obs}(w),\sigma^S_{obs}(s))}$
- a map $\sigma^F_{int(w,s)} : F_{int(w,s)} \rightarrow F'_{int(\sigma^{S*}_{int}(w),\sigma^S_{int}(s))}$
- a map $\sigma^P_{obs(w)} : Pr_{obs(w)} \rightarrow Pr'_{obs(\sigma^{S*}_{obs}(w))}$
- a map $\sigma^P_{int(w)} : Pr_{int(s)} \rightarrow Pr'_{int(\sigma^{S*}_{int}(w))}$

Models: A many sorted *Sigma*-model for a given many sorted signature $\Sigma_{obs \cup int}$ = $(S_{obs}, S_{int}, F_{obs}, F_{int}, Pr_{obs}, Pr_{int})$ consists of:

- a non empty career set s^M_o for each $s_o \in S_{obs}$

- a non empty career set s_i^M for each $s_i \in S_{int}$
- a function $f_o^M : w^M \to s^M$ for each $f_o \in F_{obs}$
- a function $f_i^M : w^M \to s^M$ for each $f_i \in F_{int}$
- a predicate pr_o^M for each $pr_o \in Pr_{obs}$
- a predicate pr_i^M for each $pr_i \in Pr_{int}$

Semantics of CSP-CASL : The semantics of this formalism is achieved in the two steps approach as given in the original article [21] of CSP-CASL specification. In the first step, evaluation is according to the data part (D_{obs}, D_{int}) where each model M of data part (D_{obs}, D_{int}) gives rise to a process denotation $(P_{obs}, P_{int})_M$. In the second step, evaluation is according to the CSP, which translates $(P_{obs}, P_{int})_M$ into chosen CSP semantics. CASL terms are used as communication and CASL sorts denote set of communications. Further, we present CSP-CASL specification of the industrial case study which is informally described into the chapter one. Throughout our thesis, we shall refer the given CSP-CASL formalism of case study to elaborate the proposed approach.

Example 1. In Table 3.1, we present CSP-CASL based formal specification of MED, which is informally described in the earlier part of this thesis. Here, the case study is abstractly specified with extended CSP-CASL syntax. At the abstract level of this specification, internal behaviors are not described, only observable behaviors are described. The CSP-CASL based formal specification starts by giving the specification name as *ComMedProtocol*, further data, channel and process part of communication protocol is described. The given formalism will be referred throughout this thesis to explain the concepts of software evolution and compliance of process improvement model.

3.3 Software refinement

In the previous section, we proposed a distinct approach to software specification. Let's see how our approach of formalism fits with existing approaches of software refinements. In this research, we propose a slightly deviated approach to software refinement. The proposed refinement technique is an extension of existing refinement techniques. The complete refinement framework is elaborated

Table 3.1: Abstract CSP-CASL specification of Basic MED

```
ccSpec ComMedProtocol
Data
        sorts Message, EncrMessage, ConnPara, Ack, DevID
        ops SerialConn: ConnPara→ Ack
            SendMessage: Message X Ack→ EncrMessage
            RecvMessage: Ack→ EncrMessage
            CloseConn:→ Ack
Channel
        EncrCh : EncrMessage
        AckCh : Ack
Process

    Send = AckCh! SerialConn(ConnPara)→
           [Message] EncrCh ! SendMessage(Message, Ack)→
           AckCh!CloseConn() → Send

    Receive = AckCh! SerialConn(ConnPara)→ [Message] EncrCh ! RecvMessage(Ack)→
           AckCh!CloseConn()→ Receive

End
```

for a CSP-CASL based specification formalism, however the proposed refinement framework can be applied to any specification formalism.

In the proposed approach of software specification, structural and behavioral properties are separately considered. Similarly, the refinement of such a formal description requires refinement of structural and behavioral properties as well. The refinement of behavioral properties is reduction in expressiveness and nondeterminism; refinement of structural properties is concerned with the replacement of data types with simpler and more implementable type. There have been various refinement frameworks for structural and behavioral specification refinement such as [10][43][14]. In this thesis, we elaborate a concept of refinement which is based on structural and behavioral specification of any software system. This refinement approach is based on the extended CSP-CASL specification language. Initially, state of art of CSP-CASL refinement is described. Subsequently, extended CSP-CASL based software refinement is presented. This approach to refinement is an extension of the existing refinement approach with a consideration of observable and internal behaviors of specified software system.

3.3.1 CASL refinement

The algebraic specification is known for an abstract specification of the software systems. The abstraction in the specification facilitates the understanding of complex requirements [39] by hiding unnecessary details at the initial steps of requirement analysis. The abstract specification subsequently evolves with the steps of refinement by fixing design details. The steps of refinement continue until specification becomes straightforward for an implementation. The specification language CASL is laced with simple refinement techniques that help to formalize the whole software development life cycle. CASL refinement technique is based on the model class inclusion [29] similar to any algebraic specification language. The refinement approach is rather based on the loose semantics of CASL specification. A refinement relation between two CASL specifications is defined as follows:

Definition 1. Given two CASL specifications Sp and $Sp^{'}$ such that $Sig[Sp] = Sig[Sp^{'}]$, $Sp^{'}$ is refinement of Sp iff $Mod[Sp^{'}] \subseteq Mod[Sp]$.

Formally, correctness of this refinement can be represented as $M_{|\sigma} \in Mod[Sp]$ for each $M \in Mod[Sp^{'}]$, where σ is a mapping of symbols from Sp to $Sp^{'}$. The simple refinement concept is a basis for CASL specification refinement; further this concept of refinement is elaborated for the CASL architectural specification in paper [69]. Our research activities are confined to the basic and structured CASL specification where CASL simple refinement is considered as a main refinement approach. Additional details of CASL refinement are described in the paper [69], particularly, CASL architectural specification is elaborated in depth. In the further subsections we describe the structural and behavioral specification refinement techniques.

3.3.2 CSP refinement

The CSP has various notions of refinements for proving equivalence between two CSP processes. The given refinement notions are based on the semantics of their specification. In denotational semantics of CSP, three well established refinement notions are *Trace*, *Failure* and *Failure-Divergence* refinement. We restrict our research activities to the *trace* and *failure* refinement notions for a CSP specification. The CSP trace and failure refinement example is graphically presented in Figure 3.3. Further details of these refinement techniques are given as follows:

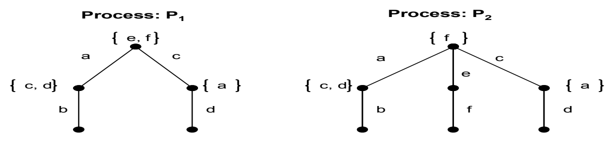


FIGURE 3.3: CSP refinements

Trace refinement

A process P_1 is a trace refinement of another process P_2 if all the traces of P_1 are in the traces of P_2. This refinement notion preserves the safety properties of a system. Formally we can write this refinement relation as follows:
$P_2 \sqsubseteq_T P_1 = traces(P_1) \subseteq traces(P_2)$

Failure refinement

Failure refinement is defined by asserting all the failures of a refined process are also failures of refining process. Formally, we can write this refinement relation for process P_1 and P_2 as follows:
$P_1 \sqsubseteq_F P_2 = failures(P_2) \subseteq failures(P_1)$

3.3.3 CSP-CASL refinement

A top-down approach of a software system development starts by formulating an abstract description, which is further refined with an addition of design details at the steps of refinement. The refinement process continues until the system specification describes all the required details for an implementation. Our approach to software refinement is based on the consideration of observable and internal behavior of the specifying software system. At the abstract level of specification only observable behaviors are specified which is further refined by adding the internal behaviors of the software system. We elaborate this concept of refinement with the formal specification language CSP-CASL in a two step approach; data refinement and process refinement. To calculate data refinement, process part of specification remains constant, and each data part induces process refinement. Similarly, to calculate process refinement, data part of specification remains constant. CSP-CASL specification refinement is defined with two steps approach:

Definition 2. Given, two CSP-CASL specifications $SP_r = (D_{robs}, D_{rint}, P_{robs}, P_{rint})$ and $SP = (D_{obs}, D_{int}, P_{obs}, P_{int})$. SP_r is a refinement of SP (represented as $SP \overset{R}{\searrow} SP_r$), iff they satisfy following $Data\ refinement$ and $Process\ refinement$ conditions:

Data refinement:

- $\Sigma(D_{obs}) = \Sigma(D_{robs})$
- $\Sigma(D_{int}) \subseteq \Sigma(D_{rint})$
- $Mod(D_{robs}\ D_{rint})_{|\Sigma(D_{obs}\ D_{int})} \subseteq Mod(D_{obs}\ D_{int})$

Process refinement:

- $traces(P_{robs}) \subseteq traces(P_{obs})$
- $traces(P_{int}) \subseteq traces(P_{rint})$
- $failures(P_{robs}) \subseteq failures(P_{obs})$
- $failures(P_{rint}) \subseteq failures(P_{int})$

Here $Mod(D_{robs}\ D_{rint})_{|\Sigma(D_{obs}\ D_{int})}$ represents the $Mod(D_{robs}\ D_{rint})$ restricted to the $Symbols(D_{obs}\ D_{int})$; such that for all $Symbols(D_{obs}\ D_{int})$ there exists an injective mapping to the $Symbols(D_{robs}\ \ D_{rint})$. In addition to the above conditions, all the axioms of $e \in Axioms(D_{obs}\ D_{int})$ must be true in the refined model of specification $Mod(D_{robs}\ D_{rint})_{|\Sigma(D_{obs}\ D_{int})} \models e$. $Symbols(x)$ represents collection of data types, operations and predicates for the selected specification context x.

Figure, 3.4 presents an overview of proposed syntax of extended CSP-CASL refinement. As a refinement of data specification, additional internal specification $dataSpec_{rint}$

ccsSpec refinedSpecName
Data
$\quad$ $\text{dataSpec}_{\text{obs}}$ and/then $\text{dataSpec}_{\text{int}}$
$\quad$ then/and
$\quad$ $\text{dataSpec}_{\text{int}}\text{R}$
[Channel
$\quad$ $\text{channelSpec}_{\text{obs}}$
$\quad$ $\text{channelSpec}_{\text{int}}$]
Process
$\quad$ $\text{processSpec}_{\text{obs}}$; $\text{processSpec}_{\text{int}}$
$\quad$; $\text{processSpec}_{\text{int}}\text{R}$
End

FIGURE 3.4: Refined CSP-CASL spefication syntax

details can be added. However, an observable data part is refined as proposed in theories of [37][43]. Similarly, the process part of specification is refined by addition of internal processes $processSpec_{rint}$. The semantics of the this refinement approach remain similar to [21].

Example 2. Here, we present a refinement of the medical case study which is given in the Table 3.1. Particularly, design decisions are incorporated into this refined specification. Additionally, axioms are declared to describe the properties of basic specification. Design decision symbols are added as internal specification. This is a simple step of refinement, which elaborates the pertinence of our proposed approach. Let us see how this refined specification is derived from our given basic specification in the Table 3.2 .

Generally, a CSP-CASL specification starts with the specification name, subsequently the required library functions are included. In the given example, for simplicity reasons, libraries are not exclusively included. Further, the structural specification is similar to the specification given in Table 3.1, where each symbol of initial specification has mapping to the symbols of refined specification. Further, the properties of specification are defined by axioms which reduces the class of models [30]. Design decision's data part is defined as internal specification. This is a special feature of our proposed approach to specification where new symbols can be added as the internal part of specification. Internal specification starts with special sign '_' which make it readable. The other part of specification is similar to the initial specification only processes are enhanced with internal design decisions properties. The complete specification is traceable with proposed definition of refinement and verifiable with our tool.

Table 3.2: Refined CSP-CASL specification of Basic MED

```
ccSpec RefinedComMedProtocol
Data
        sorts Message, EncrMessage, ConnPara, Ack, DevID
        ops SerialConn: ConnPara→ Ack
            SendMessage: Message X Ack→ EncrMessage
            RecvMessage:→ EncrMessage
            CloseConn:→ Ack
        axioms Message= Message+DevID+Ack
            EncrMessage != Message
        then _ValMsg(Message)→ Message
            _ValEncrMsg(EncrMessage)→ EncrMessage
Channel
        EncrCh : EncrMessage
        AckCh : Ack
Process

        Send = AckCh! SerialConn(ConnPara)→
               AckCh!_ValidateMsg(Message)→
               [Message]EncrCh!SendMessage(Message,Ack)→
               AckCh!CloseConn()→ Send

        Receive = AckCh! SerialConn(ConnPara)→
               [Message] EncrCh!RecvMessage()→
               _ValidateMsg(EncrMessage)→ AckCh!CloseConn()→ Receive
End
```

3.4 Software enhancement

Software enhancement is a process of adding new features to the existing software system. Additions of new features are routinely practiced in the industrial environment. But more often, they are practiced with some ad-hoc approach. There has been significant research in these aspects, such as specification enrichment, product line based software development, component based development etc. However, their industrial presence is limited to certain aspects of software enhancement. These research activities are confined to specific parts of software development. A framework for software artifact evolution parallel to software system evolution is not fully elaborated.

In this research, we elaborate a formal method based software enhancement approach for adding features to the existing software system. At the initial steps, we formally define a syntax and semantics of software enhancement. The defined concept is further investigated for software artifacts traceability and reusability. The complete understanding of the software evolution is described with the process algebraic and algebraic specification language, CSP-CASL . Initially, CASL and CSP based software enhancement is formally defined. Furthermore, this definition software enhancement is used for a proposal of software artifact enhancement and software artifact traceability. In this research, software enhancement is particularly referred to as software specification enhancement as well as software artifact enhancement.

3.4.1 CASL enhancement

Inception of algebraic specification started off as a method to formally describe abstract data type but concerns of structuring and modularity were always in research to make it applicable to the industrial practices. Industrial software systems are generally huge in nature. Developments of such systems are always based on their structuring and modularity. Here, we briefly explain these concepts of software development with the help of algebraic specification language CASL. The concept of structuring and modularity is supported by three types of software specification mechanisms in CASL. These concepts of structuring and modularity are fundamental basis for a formulation of software enhancement.

- Basic specification
- Structured specification
- Architectural specification

Algebraic specification language CASL has been developed with the adoption of various research advantages. These advantages are integrated into the features of CASL. CASL is an expressive language for a formal description of functional requirements, structural and modular software design. CASL syntax is laced with three types of extension to a CASL specification which are the basis for software enhancement. First type of extension allows to add new symbols to existing specifications by using the CASL keyword *'then'*. This type of specification extension is an integral part of basic specification. The second type of specification enhancement in CASL is supported by keyword *'and'*, which allows to unite two independent specifications generally known as structured specification. It means that enhanced features of software system can be integrated into the initial formal specification by using the keywords *'then'* and *'and'* depending on the enhancement type. An architectural specification provides the possibilities for specifying components from which larger systems can be developed. Semantics of basic and structured specification is a signature and a class of models. Architectural specification semantics represent its modular structure.

3.4.2 CSP enhancement

CSP enhancement is a concatenation of processes. There are a number of CSP operations which combine two or more CSP processes; such as a combination of processes in sequential or parallel manner. The CSP sequential composition operator allows to combine two CSP processes in sequential fashion and this combination produces a new CSP process. *';'* is used as a sequential concatenation operator. This operator can be used to facilitate modularity in the description. Suppose P and Q are the CSP processes, a sequential composition of these can be represented as $P;Q$, which means that the composed process first behaves like P until it terminates and then, it behaves like process Q. Termination of a process in CSP is represented by an operator *SKIP*. We simply assume that all processes terminate after finishing their sequence of events. There have been many research activities for deadlock and divergence analysis such as [32][23].

Another way of a CSP process enhancement is adding more CSP processes in such a way that they can interact with the original process by agreeing on some events. CSP process denotation allows to add such type of processes as parallel processes. In addition to these types, a CSP process can also be enhanced by adding new CSP processes as an external or internal choice of the existing process.

3.4.3 CSP-CASL enhancement

Software artifact reusability has been considered as an important aspect in software enhancement. However, software artifact reusability is a major challenge to achieve. In this research, we formally define a concept of software enhancement within the framework of our proposed specification technique. In this consideration, a software enhancement is a process of adding new functional or performance requirements to the existing software system by semantically preserving its existing functionalities. The formal approach to software enhancement is the foundation for software artifact reusability. The CSP-CASL based formal definition to the software enhancement is a two step approach as given below:

Definition 3. Given, two CSP-CASL specifications $SP_e = (D_{eobs}, D_{eint}, P_{eobs}, P_{eint})$ and $SP = (D_{obs}, D_{int}, P_{obs}, P_{int})$. SP_e is an enhancement of SP (represented as $SP \rightsquigarrow SP_e$), iff they satisfy following *Data enhancement* and *Process enhancement* conditions:

Data enhancement

- $\Sigma(D_{obs}) \subseteq \Sigma(D_{eobs})$
- $\Sigma(D_{int}) \subseteq \Sigma(D_{eint})$
- $Mod(D_{eobs}\ D_{eint})_{|\Sigma(D_{obs}\ D_{int})} = Mod(D_{obs}\ D_{int})$

Process enhancement

SP_e is process enhancement of SP iff $\forall\ m \in Mod(D_{eobs}\ D_{eint})$, satisfy following conditions of their traces and failures.

- $traces(P_{obs}) \subseteq traces(P_{eobs})$
- $traces(P_{int}) \subseteq traces(P_{eint})$
- $failures(P_{eobs}/P_{eint}) = failures(\Delta P_{eobs}/\Delta P_{eint}) \cup \forall f \in failures(P_{obs}/P_{int}) | f \notin traces(P_{eobs}/P_{eint})$.

To calculate data enhancement, process part of specification remains constant, and each data part induces process enhancement. Similarly, to calculate process enhancement, data part of specification remains constant. $Mod(D_{eobs}\ D_{eint})_{|\Sigma(D_{obs}\ D_{int})}$ represents the $Mod(D_{eobs}\ D_{eint})$ restricted to the $Symbols(D_{obs}\ D_{int})$; such that for all $Symbols(D_{obs}\ D_{int})$ there exists an injective mapping to the $Symbols(D_{eobs}\ \ D_{eint})$. In addition to the above conditions, all the axioms of $e \in Axioms(D_{obs}\ D_{int})$ must be true in the enhanced model of specification $Mod(D_{eobs}\ D_{eint})_{|\Sigma(D_{obs}\ D_{int})} \models e$. $Symbols(x)$ represents collection of data types, operations and predicates for the selected specification context x. Process enhancement is an addition of new traces into observable or internal part of specification. A software system is an enhancement of earlier version, if it has more observable traces and all the failures from previous version are either failures of enhanced version or they are into observable traces of enhanced version. All the failures of an enhanced software system are the union of new $failures(\Delta P_{eobs}/\Delta P_{eint})$ which are specified as an enhancement into specification and all the failures from previous version which are not into observable traces of enhanced specification. Further, the proposed theoretical concepts are elaborated with our industrial case study, initial version of MED has specified that MED is only allowed to connect with the serial interface of a computer, all other types of connection should not be permitted e.g Ethernet based connection was not permitted. In initial MED, ethernet based connection was into failures of specification however, in enhanced version of MED this became a new feature.

ccsSpec enhancedSpecName
Data
 dataSpec$_{obs}$ and/then dataSpec$_{int}$
 then/and
 dataSpec$_{obs}$E and/then dataSpec$_{int}$E
[Channel
 channelSpec$_{obs}$
 channelSpec$_{int}$]
Process
 processSpec$_{obs}$; processSpec$_{int}$
 ; processSpec$_{obs}$E ; processSpec$_{int}$E
End

Figure 3.5: Enhanced CSP-CASL specification syntax

Figure 3.5 presents a visual view of our definition of software enhancement. A software enhancement allows to add new data and process part to the existing specification.

Example 3. Here in Table 3.3, we present an enhanced version of the medical case study which is given in the Table 3.1. Particularly, the enhanced version of MED is advanced with various types of connection possibilities such as Ethernet and dial up connection. A CSP-CASL based formal specification to this enhanced version of MED is given in the Table 3.3. Let us see how this enhanced specification is related to given basic specification in the Table 3.1 .

With this enhancement definition, we can show a relation between these two specifications. To start with, we define the map between basic specification and enhanced specification. Clearly, enhanced specification has mapping for all the basic specification sorts except two new functions *EthernetConn* and *DialupConn*. Signature of enhanced specification has grown with a conservative extension, such that model of enhanced specification with the restricted signature to basic specification is equal to the model of basic specification. As the process part, basic specification has only two process, *Send* and *Receive*, these processes are also part of enhanced specification. However enhanced specification traces are enlarged with traces of *EthernetConn* and *DialupConn* related connections. Number of traces in enhanced specification are more than number of specification into basic specification. All the traces and failures of basic specification are also part of enhanced specification. All the conditions of enhancement definition are true, thus we can prove that CSP-CASL specification given in Table 3.3 is an enhancement of CSP-CASL specification given in Table 3.1.

TABLE 3.3: Abstract CSP-CASL specification of Enhanced MED

```
ccSpec EnhancedComMedProtocol
Data
        sorts Message, EncrMessage, ConnPara, Ack, DevID
        ops SerialConn: ConnPara→ Ack
            SendMessage: Message X Ack→ EncrMessage
            RecvMessage:→ EncrMessage
            CloseConn:→ Ack
        then EthernetConn: ConnPara→ Ack
            DialupConn: ConnPara→ Ack
Channel
        EncrCh : EncrMessage
        AckCh : Ack
Process

        Send=AckCh!SerialConn(ConnPara)→
            [Message] EncrCh!SendMessage(Message,Ack)→
            AckCh!CloseConn()→ Send
           []
            AckCh!EthernetConn(ConnPara)→
            [Message] EncrCh!SendMessage(Message,Ack)→
            AckCh!CloseConn()→ Send
           []
            MsgCh!DialupConn(ConnPara)→
            [Message] EncrCh!SendMessage(Message,Ack)→
            AckCh!CloseConn()→ Send

        Receive = MsgCh!SerialConn(ConnPara)→ [Message] EncrCh!RecvMessage()→
            AckCh!CloseConn()?Receive
           []
            MsgCh!EthernetConn(ConnPara)→[Message] EncrCh!RecvMessage()→
            AckCh!CloseConn()→ Receive
           []
            AckCh!DialupConn(ConnPara)→ [Message]EncrCh!RecvMessage()→
            AckCh!CloseConn()→ Receive

End
```

3.4.4 Enhancement through extension

In this approach of software enhancement formalization, new user requirements are integrated into the existing software system. The integration of new user requirement to the existing software specification is supported by formal specification syntax. The CSP-CASL based formal definition to this type of software enhancement is similar to the given definition of software enhancement, specially distinguished by the keyword "*then*" of data part specification. Process part enhancement approach is similar to as proposed into the definition of software enhancement.

3.4.5 Enhancement through substitution

This approach of software enhancement allows to add independently developed new functional or performance requirements into the existing software system. Typically, this type of enhancement is a component based software enhancement. Here, new functional and performance requirements are separately developed and later integrated as a component. The above CSP-CASL based formalism supports this type of enhancement with the CASL keyword *and*, CSP keyword ; by allowing the integration with the existing software specification.

3.4.6 Enhancement through extension and substitution

This approach of software enhancement allows to extend functional or performance requirements by extension and substitution type of software enhancement as defined in the above subsections. A formalism of this type of enhancement is the integration of the above two type of software enhancements. Syntax and semantics can easily be interpreted with the integration of the above two specification formalisms.

3.4.7 Enhancement as software product line

The product line approach is based on a set of common features. Each element of the product line contains some common features; they are used to identify products as belonging to the same line. The common features are preserved and enhanced in the development of any product of this line. Commonality of the features has various significant advantages. Besides providing a familiar look-and-feel to the end customer, it allows a systematic reuse of artifacts in the development process. This can signicantly improve quality and cost efficiency. The CSP-CASL based software enhancement definition allows to investigate product line based software development. Particularly, the definition of *enhancement through substitution* and *enhancement through extension* allow to elaborate concepts of product line based software development.

3.5 Summary

In this chapter, a distinct formal specification technique is introduced which allows to distinguish observable and internal behaviors of a specifying software system. The proposed specification technique provides a pragmatic approach for the description of abstract and detailed specification of a software system. Generally, at the initial steps of software specification only abstract behaviors are specified. Subsequently, the abstract specification is evolved into detailed design by fixing design details. The approach of fixing design details is considered as an addition of internal behaviors. In this research, the CSP-CASL based syntax and semantics is presented to describe a software system specification as observable and internal behaviors.

This proposed approach of software specification is further investigated in vertical and horizontal evolution of the software systems In vertical software evolution, the established idea of software refinement is extended with the consideration of observable and internal behaviors, CSP-CASL based software refinement is formally described and elaborated with our proposed case study of medical embedded device. Further, as a horizontal software evolution, the idea of software enhancement is formulated with extended CSP-CASL syntax and semantics. The defined concept is investigated for software artifacts traceability and reusability. Software enhancement techniques are categorized to tackle the needs of today's industrial environment.

These formulations of software specification, software refinement and software enhancement play a key role to formalize the concepts of software system evolution. In further chapters these definitions are taken as a foundation for product and process quality improvement framework development.

Chapter 4

Product quality framework

Software product quality planning should start with an unambiguous, precise and complete description of user requirements. Requirement gathering is the most important step for any software product development. The quality of a software system is hugely dependents on the requirement specification. Software product quality values vary with the criticality and applicability of software products. In safety critical systems, software quality has a different meaning than a simple business processing application e.g. (Mobile communication, Enterprise Resource Planning, Business IT Management etc.). A lack of quality in a safety critical system might lead to an unrecoverable loss. However a quality problem in a simple business processing application might be tolerable by its users. In the development of software systems, quality requirements are decided with various business and operational aspects in mind. In general, software system quality is assured by the combination of product and process quality. Product quality is closely related to the conformance of the user requirement in terms of different types of testing and verification techniques. However, process quality is more concerned with defect prevention and efficient software life cycle management. A good process is generally achieved with the support of standard process models such as CMMI, ISO, SPICE, V-model$_{xt}$, Six sigma etc.

Our proposed product quality framework is established on the foundation of formal method based software development and software evolution. Particularly, we describe a distinct approach to test case derivation and maintenance based on extended CSP-CASL formalism. In the subsequent sections of this chapter, we present a detailed view on the proposed software product quality framework. At the end of this chapter, a brief description of the developed CSP-CASL evolution tool (*cc-FormTest*) is presented. The developed tool provides pragmatic support to our proposed software quality framework. The proposed software quality framework is further explained by the development and maintenance of the selected medical embedded system case study.

4.1 Testing terminologies

Formal methods based software system development has shown significant benefits in the software verification and validation activities [41] [70] [18]. In this research, we investigate the advantages of CSP-CASL based software system formalism in testing domain. The proposed testing terminologies are built on the existing research proposed by [71] [63] [72] [48]. Particularly, we have extended test case derivation, test case maintenance and test case evaluation methodologies of the existing research of our group [71]. In this paper, deep theoretical approach of software refinement and testing is presented for a CSP-CASL based formalism. In particular, for a test case the test oracle and test evaluation problem is separated with a definition of expected result (green, red and yellow) and the test verdicts (pass, fail and inconclusive). These concepts allow specification transformation, refinement and test execution at all stages of software system's design [71]. In this thesis, we have considered more pragmatic approach to specification refinement and test generation and test evaluation. The proposed framework is developed with a consideration of product and process quality improvement.

4.1.1 Test case

A major challenge in a software testing is the selection of appropriate test cases. Generally, functional requirements are the starting point to derive test cases either manually or automatically. However, only functional requirement based testing is not enough to guarantee the behaviors. The software systems are required to respond properly for expected as well as for unexpected behaviors. This type of desire has extended a need of testing for expected behaviors as well as for unexpected behaviors. In this consideration, we extend the understanding of software testing with a direction of positive and negative test case generation. Positive test case generation has been discussed in many research papers such as [72][73].

Negative test case generations are still at an initial stage. Generally, they are practiced in industrial environments; however, they are explored less with different specification approaches.

Proposed test generation mechanism is based on the CSP-CASL formalism of a software system. For a CSP-CASL specification, we propose test cases generation for expected behaviors (traces) as well as for the restricted behaviors (failures) of the considered software system. Each test case confirms the expected or restricted feature of the described CSP-CASL specification. A test case which confirms an expected behavior is referred to as a **Positive test case** and a test case which confirms a restricted behavior of is referred to as a **Negative test case**.

Definition 1. A *test case* is a CSP-CASL trace or failure.

As previously defined that for a CSP-CASL specification formalism, CSP-CASL traces are categorised as **Complete trace** and **Incomplete trace**. Similarly, **Positive test cases**(T_p) can be categorized as **Complete positive test cases**(T_c) and **Incomplete positive test cases**(T_i) such as $T_p = T_c \cup T_i$. A positive complete test case is a trace of the expected behavior where each event of the trace terminates successfully [32]. A positive incomplete test case is a trace of the expected behavior where last event of the trace does not terminate successfully [32] . The incomplete positive test case is used to debug the location of a bug in a software system. If a software system passes a positive complete test case, then it also passes all the incomplete test cases which are the subset of this positive complete test case.

Definition 2. A *test suite* is a set of test cases.

This approach of test generation allows to test software system for the intended as well as unintended behaviors. Modern programming languages are prepared to handle intentional and unintentional behaviors with a mechanism of exception handling. This methodology of test generation allows testing of intended behavior as well as testing of behaviors which are supposed to throw exceptions. Specification of failures has been in discussion since the proposal of CSP [32]. However the testing methodologies of such behaviors are not considered in the researches. Below, we present an example to elaborate the above concepts of test generation for a CSP-CASL specification.

Example 1. Let there be a CSP-CASL process P_1 with the traces $(\langle a, b\rangle\ \{a, d\})$ and $(\langle a, b, c\rangle\ \{e\})$. The above definition of test case allows to derive following test cases for the process P_1. Few positive test cases derived from the process P_1 are $\langle a, b\rangle, \langle a, b, c\rangle$. Similarly, negative test cases derived from the process P_1 are $\langle a, b, a\rangle, \langle a, b, d\rangle, \langle a, b, c, e\rangle$.

Generally a CSP-CASL specification has infinite traces and failures. Test selection strategy is a challenging area of research, in this research test selection strategy is considered out of scope. However, few selection strategies are considered from research papers such as [19] [74]. The negative test case generation is more challenging than the positive test case. In our consideration, we have extended the specification mechanism which allows to write the unintended behaviors as the failures of specification. This part of specification allows to derive negative test cases. The evaluation of test case execution on the specification or on the implementation is the next required step for testing. This is discussed in the further subsections.

4.1.2 Test verdict

Test verdict is assessment of correctness of specification/implementation against a test case. The executable specification or implementation is collection of all possible processes of the considered CSP-CASL specification. The executable specification or implementation is referred as SUT. Internal details of a SUT is hidden, however any input and output can be accessed from outside. A test case execution on the executable specification or implementation is like parallel processes (Test Case || SUT), such that they agree on everything. Test case is assumed to be derived with expected or restricted behavior. Test case evaluation is a time consuming process, so it is always desirable to test a software system against the test cases which are good enough to verify the properties of a software system. The test verdict of a test case T on a given CSP-CASL specification $Sp = (D_{obs}, D_{int}, P_{obs}, P_{int})$ is a value among $\{Pass, Fail, Inconclusive\}$. Test verdict is assigned with following conditions:

- **Pass:**
  - $(\Sigma(T) \subseteq \Sigma(Sp)) \wedge (Mod(T) = Mod(Sp)_{|\Sigma(T)})$
  - iff for all $m \in Mod(Sp) : T_m \in traces(Sp)_m$
- **Fail:**

- $(\Sigma(T) \subseteq \Sigma(Sp)) \wedge (Mod(T) = Mod(Sp)_{|\Sigma(T)})$
- iff for all $m \in Mod(Sp) : T_m \in failures(Sp)_m$

- **Inconclusive**(Inc):
  - Not in the above two conditions.

Interpretation of a test case against the execution of specification/implementation is known as test oracle. Generally, test oracle is some type of comparison of actual output for some input against calculated output for the same input. As the formal specification is a complete and precise description of the system, it is easy to calculate the expected output for a given input. The known issue with algebraic specification is non-determinism in the specification. Since non-determinism is not interesting from a practical point of view. In our research, we only consider deterministic specification, non-determinism is considered out of scope for this research.

These definitions of a test case and test verdict are further investigated in the evolution of software systems. In further sections, we investigate the reusability of test cases in the horizontal and vertical evolution of software systems. The test case reusability is investigated with the persistence of test verdicts from the initial specification to evolved specification. To explain testing life cycle, we start with the derivation of test cases from the proposed case study and further, these test cases are elaborated into the evolution of software system. Below, we present few test cases and their verdicts from our proposed case study of MED. At the initiation steps, we consider only functional test cases which are based on the given observable CSP-CASL specification. The signature of a test case is a subset of the signature of specification (test case definition). Once we execute these test cases on the specification or on the implementation, we categorize the results of execution according to proposed test verdict definition. This is decided with the help of a procedure referred as test oracle [50].

Example 2. Here, we present a set of test cases derived from the CSP-CASL specification of MED as given in Chap-3 Table 3.1. These test cases are presented in the Table 4.1, each test case is either a trace(Positive test case) or a failure(Negative test case) of $ComMedProtocol$ within the given signature. The expected test verdict of these test cases is determined by applying the definition of test verdict on the $ComMedProtocol$ CSP-CASL specification. The test case Tc_1 is in the traces

Table 4.1: Abstract Medical Device Test cases and expected test verdict

Tc_1	$\langle SerialConn(ConnPara)\rangle$	Pass
Tc_2	$\langle SerialConn(ConnPara) \rightarrow SendMessage(Message)\rangle$	Pass
Tc_3	$\langle SerialConn(ConnPara) \rightarrow SendMessage(Message) \rightarrow CloseConn() \rightarrow Send\rangle$	Pass
Tc_4	$\langle SerialConn(ConnPara) \rightarrow Message\rangle$	Fail
Tc_5	$\langle SerialConn(ConnPara) \rightarrow SerialConn(ConnPara)\rangle$	Inc

of $ComMedProtocol$ within the signature of specification. From the definition of test verdict, an expected test verdict of this test case is $Pass$. In similar fashion, Tc_2 and Tc_3 are in the traces of $ComMedProtocol$ (test verdict $Pass$). The Tc_4 has the last event as $Message$, which is in the failures of $ComMedProtocol$; this produces a test verdict for this test case as $Fail$. Further, test case Tc_5 is neither in the traces nor in the failures of $ComMedProtocol$ which leads to an expected test verdict as Inc. In subsequent sections, these test cases are investigated to elaborate the reusability into the evolution of MED.

In practice, execution of a test case is a time consuming and costly activity. Testing of software system for the test cases which do not detect any bug should be avoided. The test cases which return test verdict as Inc are not actually contributing for the testing of software system. These test cases are derived at the abstract level of specification. They are considered to be evolved at a refined or enhanced level of specification. These test cases are categorized with unusable test cases before starting actual testing to save testing effort and cost.

Definition 3. A test case is unusable for the testing of a specification or its implementation if the test verdict of this test case is Inc.

4.2 Vertical software evolution and test case reuse

Generally, development of a software system starts with an abstract specification of desired properties. The abstract specification is further refined into the implementation by fixing design decisions. There is always a chance to introduce new errors within the steps of abstract to subsequent levels of specification. The error detection time and cost grows exponentially if they are not disclosed at the early stage of software development [75] [76]. Early error detection is possible by starting testing activities at very early phases of software system development. Overall software system development time and cost can be minimized by starting testing activities at the early phases of software development. However, this brings a new challenge to reuse the test cases of abstract specification into the refined specifications. In this thesis, we present a pragmatic investigation into this challenge. The complete methodology is based on the software system formalism into CSP-CASL . In further subsections, we describe our proposed approach of software refinement and testing.

4.2.1 Software refinement and testing theory

Software refinement allows a traceability of software artifacts from abstract specification to a detailed specification for a software system development. However, the relationship between abstract specification test cases and detailed specification test cases is still not fully understood. There have been many research articles about a reusability of the test cases [77] [74] however, the concept of test case reusability by software refinement is not fully practiced. This research activity of software refinement and test reuse methodology is a distinct approach from the existing research. Formally, we define the proposed testing methodology for a CSP-CASL based specification and test derivation as follows:

Definition 4. A test case is reusable at the refined specification by applying the similar steps of refinement on specification as well as on the test case. Test cases generated from an abstract specification should incorporate the refinement properties to test the refined specification.

Let's assume, an abstract specification Sp_{abs} and test suite Ts, such that Ts is derived from specification Sp_{abs}. If Sp_{abs} is refined to Sp_{ref} by applying Δr refinement step then Ts should also be refined with Δr step. Here, Δr is assumed to be a CSP-CASL refinement step. In further sections, we investigate this relationship based on the given definitions of software refinement and testing terminologies.

Theorem 1. Given a CSP-CASL abstract specification Sp_{abs} and its refined specification Sp_{ref}, Ts is test suite derived from Sp_{abs} for the testing of Sp_{abs}. If Ts' is test suite for the testing of Sp_{ref} then Ts' must be a refinement of Ts.

Proof. Our goal is to prove the refinement relation $Ts \searrow Ts'$. To prove this we are required to establish the following relation(from the definition of CSP-CASL based software refinement and test case):

- $\Sigma(Ts) = \Sigma(Ts')$
- $Mod(Ts') \subseteq Mod(Ts)$
- $traces(Ts') \subseteq traces(Ts)$
- $failures(Ts') \subseteq failures(Ts)$

Given Sp_{ref} is a refinement of Sp_{abs}, from the definition of refinement we can write:

- $\Sigma(Sp_{ref}) = \Sigma(Sp_{abs})$
- $Mod(Sp_{ref}) \subseteq Mod(Sp_{abs})$
- $traces(Sp_{ref}) \subseteq traces(Sp_{abs})$
- $failures(Sp_{ref}) \subseteq failures(Sp_{abs})$

Let's assume that the given specifications Sp_{abs} and Sp_{ref} have only test cases T and T' respectively. From the definition of test case and the refinement relation, we prove the required condition for T and T' by replacing Sp_{abs} and Sp_{ref} respectively. Similarly with induction, we prove this for all the test cases of the given test suite, which satisfies the required condition. Figure 4.1 represents a graphical view of this theorem. Here, T is a test suite for the testing of initial specification SP_I and SP_{R1} is a refined specification of SP_I, therefore all the test cases of T can be used for the testing of SP_{R1} iff they are refined with similar steps of refinement as SP_{R1}. $\square$

Theorem 2. Refinement of abstract test cases leads to the categorization of test cases into reusable or unusable.

Proof. Suppose, Sp_{abs} and Sp_{ref} are given abstract and refined CSP-CASL specifications. From the definition of refinement, the specifications Sp_{abs} and Sp_{ref} satisfy the following relation:

- $\Sigma(Sp_{ref}) = \Sigma(Sp_{abs})$
- $Mod(Sp_{ref}) \subseteq Mod(Sp_{abs})$
- $traces(Sp_{ref}) \subseteq traces(Sp_{abs})$

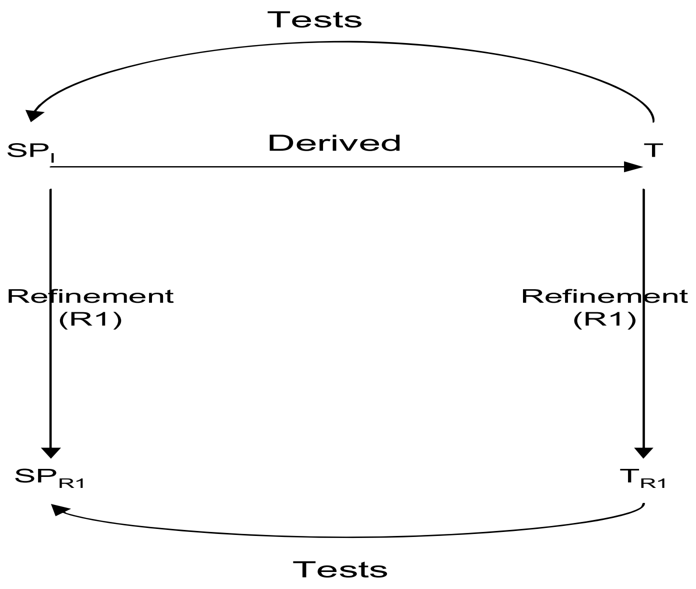


FIGURE 4.1: Test refinement and Specification refinement

- $failures(Sp_{ref}) \subseteq failures(Sp_{abs})$

From the definition of test case, each $traces$ and $failures$ represent a test case for a CSP-CASL specification. Let us assume, positive and negative test suites derived from Sp_{abs} are TS_p and TS_n. From the definition of refinement relation between Sp_{abs} and Sp_{ref}; $\exists t \in traces(Sp_{abs}) : t \notin traces(Sp_{ref})$. This proves that all the positive test cases are not the test cases for refined specification. A test case t has test verdict as $''Pass''$ iff $t \in traces(Sp_{abs})$. If the test case is not in the $traces(Sp_{abs})$, this might lead to the test verdict as Inc. Similarly we can prove for negative test cases. Subsequently, we can say that refinement of test cases saves testing effort by the categorization of test cases into reusable and unusable. $\square$

This theorem is the basis for deciding reusability of test cases, since an abstract specification allows to select infinite number of test cases. These test cases must be categorized as reusable or unusable, otherwise running unusable test cases on an implementation is a waste of resource and time. Test case refinement will allow to select only appropriate test cases which are required to test the refined specification. Refinement properties such as model class of inclusion, data type selection, non-determinism will allow to categorize test cases as reusable and unusable for

$$\begin{tikzcd}
SP_I \arrow[r, "Derived"] \arrow[d, "Refinement (R1)"'] & T \arrow[l, bend right, "Tests"'] \arrow[d, "Refinement (R1)"] \\
SP_{R1} & T_{R1} \arrow[l, bend left, "Tests"]
\end{tikzcd}$$

refined specification. In Table 4.2, we present the test cases and their test verdict from the refined level of case study $RefinedComMedProtocol$.

Table 4.2: Refined Test cases and expected test verdict from Refined Specification

Tc_1	$\langle SerialConn(ConnPara)\rangle$	Pass
Tc_2Ref	$\langle SerialConn(ConnPara) \rightarrow \_ValidateMsg(Message) \rightarrow SendMessage(Message)\rangle$	Pass
Tc_3Ref	$\langle SerialConn(ConnPara) \rightarrow \_ValidateMsg(Message) \rightarrow Message\rangle$	Fail
Tc_4Ref	$\langle SerialConn(ConnPara) \rightarrow \_ValidateMsg(Message) \rightarrow SendMessage(Message) \rightarrow CloseConn() \rightarrow Send\rangle$	Pass
Tc_5	$\langle SerialConn(ConnPara) \rightarrow SerialConn(ConnPara)\rangle$	Inc

Table 4.2 presents the refined test cases of Table 4.1. The refined test case names are postfixed with a Ref. The test case Tc_1 is directly reusable from abstract specification to a refined specification and the test verdict is also maintained. Test cases Tc_2 to Tc_4 are refined according to specification refinement so that these test cases are reusable at the refined specification. The steps of refinement are similar to the test cases as well as to the abstract specification. These test cases preserve their test verdict from abstract to refined specification. The abstract test case of Tc_3 has test verdict as $'Pass'$ in abstract specification. Here, test case Tc_3Ref is refined according to specification refinement steps and produces test verdict $'Pass'$. The remaining two test cases preserve their properties from an abstract model to a refined specification model.

Figure 4.2 represents a test suite reusability framework. This diagram shows abstract CSP-CASL specification, refined CSP-CASL specification and test suite derived from abstract CSP-CASL specification. Subsequently, the test suite is analyzed for the testing of refined specification and reusability analysis is presented. This framework is based on theoretical definitions of testing terminologies, and its implementation is supported by our developed tool. The framework gives an overview of test suite reusability analysis by categorization of test cases. As the name suggests that, reusable test suite can be used to test refined specification. Unusable test suite is automatically separated from the test suite. This approach allows to consider only reusable test cases for the testing of refined specification. The presented framework is partially implemented in our proposed testing tool, this tool is briefly described at the end of this chapter.

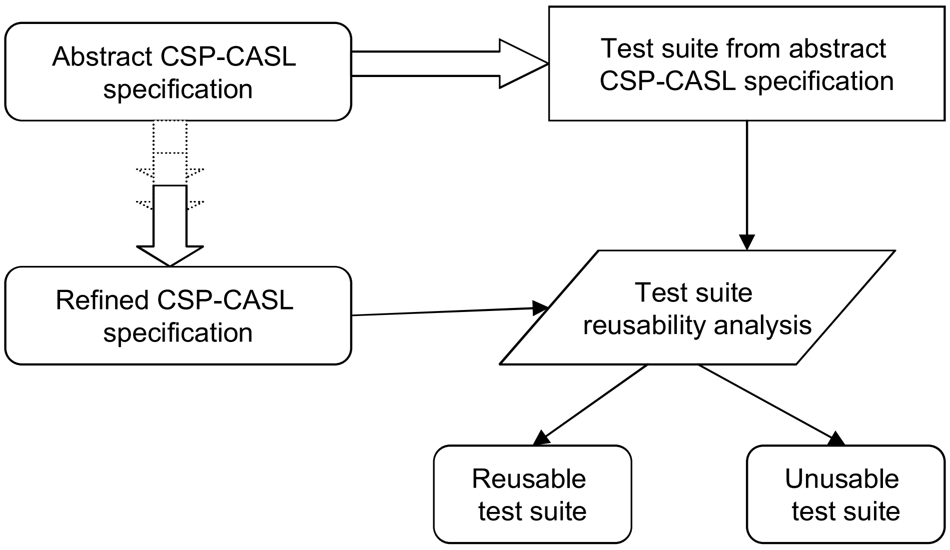

Abstract CSP-CASL specification
Test suite from abstract CSP-CASL specification
Refined CSP-CASL specification
Test suite reusability analysis
Reusable test suite
Unusable test suite

FIGURE 4.2: Test suite reusability framework for refined specification

4.3 Horizontal software evolution and test case reuse

Horizontal software evolution is a process of adding new features to the existing software system by semantically preserving its existing features. Horizontal software evolution is precisely referred to as software enhancement. Horizontal software evolution is based on the common behaviors which are maintained at subsequent levels of software evolution. Particularly, we investigate the possibilities to reuse test aspects from initial software system to enhanced software systems. In this thesis, the CSP-CASL based formal definition of software enhancement is investigated into the testing domain. In further subsections we elaborate our approach of software enhancement, where previously given formal definitions of software enhancement and testing terminologies are used as foundation.

4.3.1 Software enhancement and testing theory

The previously defined formal definition of software enhancement allows to establish a relationship between the test cases of initial and enhanced specifications.

The formal definition of software enhancement is basis for the following definition of test reusability.

Definition 5. A test case is reusable at the enhanced specification by applying similar steps of enhancement on specification as well as on the test case. Test cases derived from an initial specification should incorporate the properties of enhanced features to test the enhanced specification.

This definition of test reusability allows to establish various testing aspects for software enhancement. The suggested definition is the basis for the proposal of test reusability framework for various types of software enhancement such as product line and component based software development.

Theorem 3. Let Sp_i and Sp_e be initial and enhanced CSP-CASL specifications. A test suite TS is derived from Sp_i to test specification Sp_i then TS is also a test suite for the testing of enhanced specification Sp_e

Proof. Given Ts is derived from the given CSP-CASL Sp_i
$\Rightarrow \forall\ t : TS\ t \in traces(Sp_i)\ or\ t \in failures(Sp_i)$
Since Sp_e is enhancement of Sp_i so from the definition of enhancement we can write
$\forall\ t : TS : t \in traces(Sp_e)\ or\ t \in failures(Sp_e)$
$\Rightarrow TS$ is also a test suite for the testing of enhanced specification Sp_e. □

Theorem 4. A CSP-CASL based *complete test case* derived from an initial CSP-CASL specification can be reused at the enhanced CSP-CASL specification iff *complete test case* and specification are enhanced with similar steps of enhancement.

Proof. Let T_c be a complete test case for a CSP-CASL specification Sp_i such that T_c is in the complete traces of Sp_i
$\Rightarrow T_c$ leads to a test verdict of either $Pass$ or $Fail$
Let Sp_e be enhancement of Sp_i, such that trace of Sp_i has been added with a new event. In this case T_c is a valid trace of enhanced specification Sp_e but this is not a complete trace. By applying similar steps of enhancement on test cases, the test case can be reused for the testing of enhanced specification. □

Theorem 5. Let Sp_i and Sp_e be the initial and enhanced CSP-CASL specifications. A CSP-CASL based positive test suite TS_p preserves its test verdict from Sp_i to Sp_e.

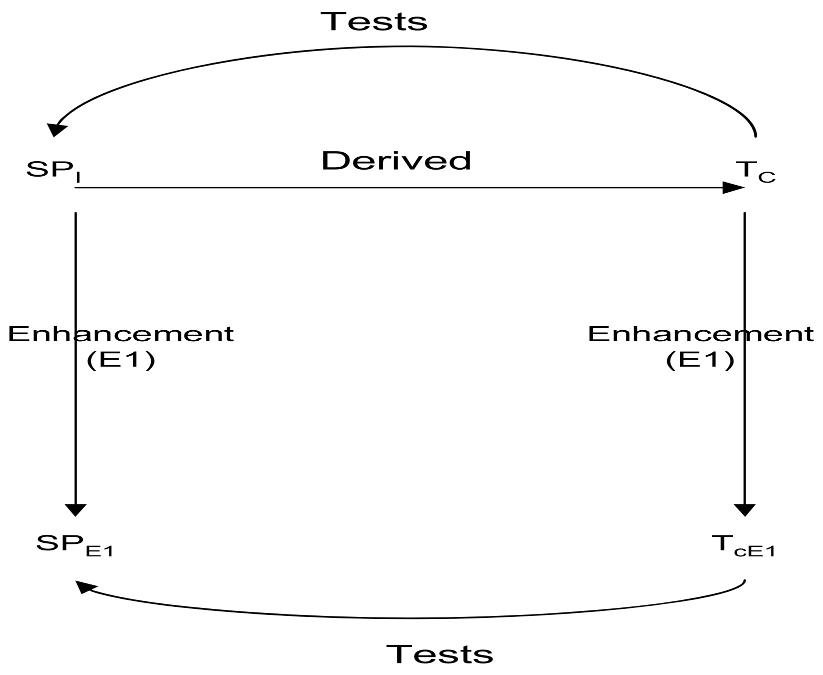


FIGURE 4.3: Test enhancement and Specification enhancement

Proof. Given, an initial CSP-CASL specification Sp_i and a test suite TS_p such that $\forall t \in TS_p$ has test verdict *Pass* in the testing of Sp_i.
$\Rightarrow \forall t : TS_p : t \in traces(Sp_i)$ and $t \notin failures(Sp_i)$ [definition of test case]
Given, Sp_e is enhancement of Sp_i, from the definition of enhancement we can write
$\Rightarrow \forall t : TS_p : t \in traces(Sp_e)$ and $t \notin failures(Sp_e)$
$\Rightarrow$ test verdict is preserved from initial to enhanced specification. $\square$

The above theorem proves the preservation of positive test cases in the enhancement of a software system. Further in the Table 4.3, we present analysis on the test cases derived from abstract MED. Especially, similar enhancement steps are applied on the test cases to achieve reusability of test cases for the testing of enhanced specification. The test verdict for each test case is presented by applying the definition of testing terminologies for enhanced specification and test case.

The definition of the enhancement relation allows us to prove equivalence between the initial and the enhanced MED. But this equivalence is achieved by restricting the signature of enhanced specification to the abstract specification. For the testing of enhanced MED, all of its properties have to be considered. To achieve this, the abstract test cases have to be enhanced with restricted properties of MED.

Table 4.3: Test cases from Abstract Medical Device and expected test verdict from Enhanced Specification

Tc_1	$\langle SerialConn(ConnPara)\rangle$	Pass
Tc_2	$\langle SerialConn(ConnPara) \rightarrow SendMessage(Message)\rangle$	Pass
Tc_3	$\langle SerialConn(ConnPara) \rightarrow SendMessage(Message) \rightarrow Message\rangle$	Fail
Tc_4	$\langle SerialConn(ConnPara) \rightarrow SendMessage(Message) \rightarrow CloseConn() \rightarrow Send\rangle$	Fail
Tc_5	$\langle SerialConn(ConnPara) \rightarrow SerialConn(ConnPara)\rangle$	Inc

The Figure, 4.4 provides a test reusability framework for the testing of enhanced specification. This framework allows to categorize test suite derived from initial specification into reusable or unusable test suite. Reusable test suite can be used for the testing of enhanced specification. The proposed framework is applied for the development of our proposed testing analysis tool. The details of this tool are presented in the further section.

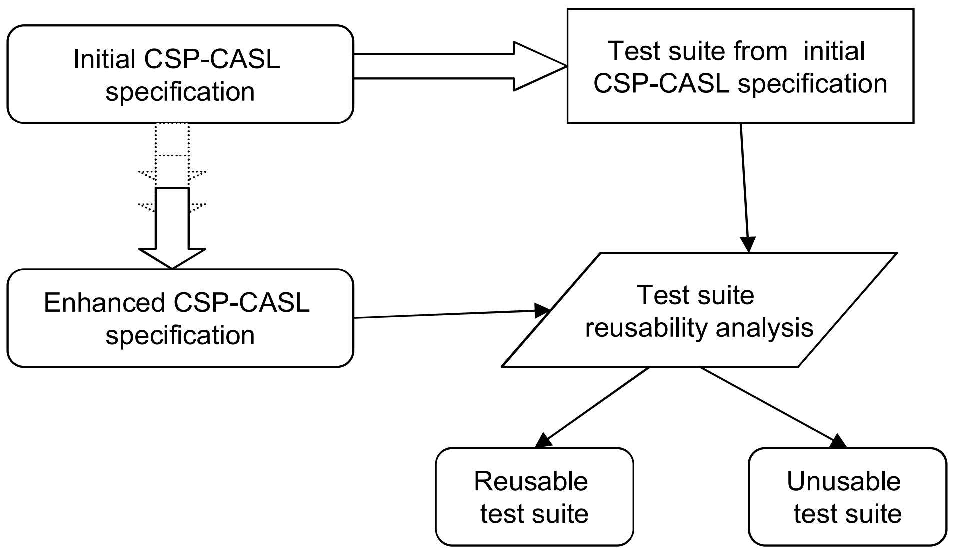

Initial CSP-CASL specification
Test suite from initial CSP-CASL specification
Enhanced CSP-CASL specification
Test suite reusability analysis
Reusable test suite
Unusable test suite

Figure 4.4: Test suite reusability framework for enhanced specification

4.4 Summary

Today, software artifact reusability is considered an important software development paradigm. The software artifact reusability allows software companies to improve time to market, cost, productivity and quality. Particularly, we investigated testing concepts for the CSP-CASL based specification formalism. Previously defined formalisms of software specification, software refinement and software enhancement are used to propose testing terminologies in the vertical and horizontal software development paradigm.

The software systems are required to respond appropriately for expected as well as for unexpected behaviors. This type of desire has extended a need of testing for expected behaviors as well as for unexpected behaviors. With this consideration, we extend the understanding of software testing with a direction of positive and negative test case generation. The proposed test generation mechanism is based on the CSP-CASL formalism of a software system. This research presents a distinct approach to test generation and test verdict interpretation during the evolution of software systems. In the following table, we present the test verdict interpretation from abstract to refined specification and from initial to enhanced specification.

Table 4.4: Test verdict transition from Abstract to Refined Specification

Abstract specification	**Refined specification**
Pass	Pass
Pass	Inc
Fail	Fail
Fail	Inc
Inc	Inc

Table 4.5: Test verdict transition from Initial to Enhanced Specification

Initial specification	**Enhanced specification**
Pass	Pass
Fail	Pass
Fail	Fail
Inc	Pass
Inc	Fail
Inc	Inc

We proposed an approach to interpret the transformation of the test verdict in the refined and enhanced model of the specification. This approach is based on the formalism of CSP-CASL . Subsequently, based on given definitions test artifact reusability theorems are proved. The details given in Table 4.4 and 4.5 are based on the given definitions of test artifacts and proposed theorems . Further, the formalism is enhanced to understand the test suite reusability by test suite categorization into reusable and unusable test suites. A simple categorization framework is presented for specification refinement and specification enhancement. This framework helps to automate the categorization of test cases. This categorization will help to decide reusability of test cases for the testing of refined and enhanced software systems. Refinement of a test suite will also allow to split the test cases in such a way that they can be directly used for the testing of the implementation.

4.5 CSP-CASL specification evolution and testing tool: ccFormTest

Formalism of the software system specification allows to automate various testing processes such as test generation, test evaluation. Only few formal specification based testing tools have been proposed to support software evolution and testing process relation. However, software evolution has been very commonly practiced in the vertical and horizontal software system development and maintenance. Our theoretical research concept is practically elaborated with a CSP-CASL specification and test case evolution tool support. This tool allows to analyze CSP-CASL based specification and elaborates a possibility of test suite reusability. Test suite reusability is especially elaborated in the software evolution. An approach of positive and negative test generation is elaborated based on our theoretical definitions. Further, we present tool architecture and its functionalities in the further subsections.

4.5.1 Tool architecture: ccFormTest

ccFormTest architecture is presented in Figure 4.5. The complete architecture is divided into various units of processing. Input units of *ccFormTest* accepts specifications and external test suite. These units are represented as A,B,C in the Figure 4.5, units A and B accept CSP-CASL specification as an input for specification analysis, test case generation and evaluation. Unit C allows to accept external test cases which can be evaluated on the specifications provided from units A and B. Further inputs of units A,B,C are syntactically and semantically validated into the unit D. Units E and F analyze the input specifications provided from units

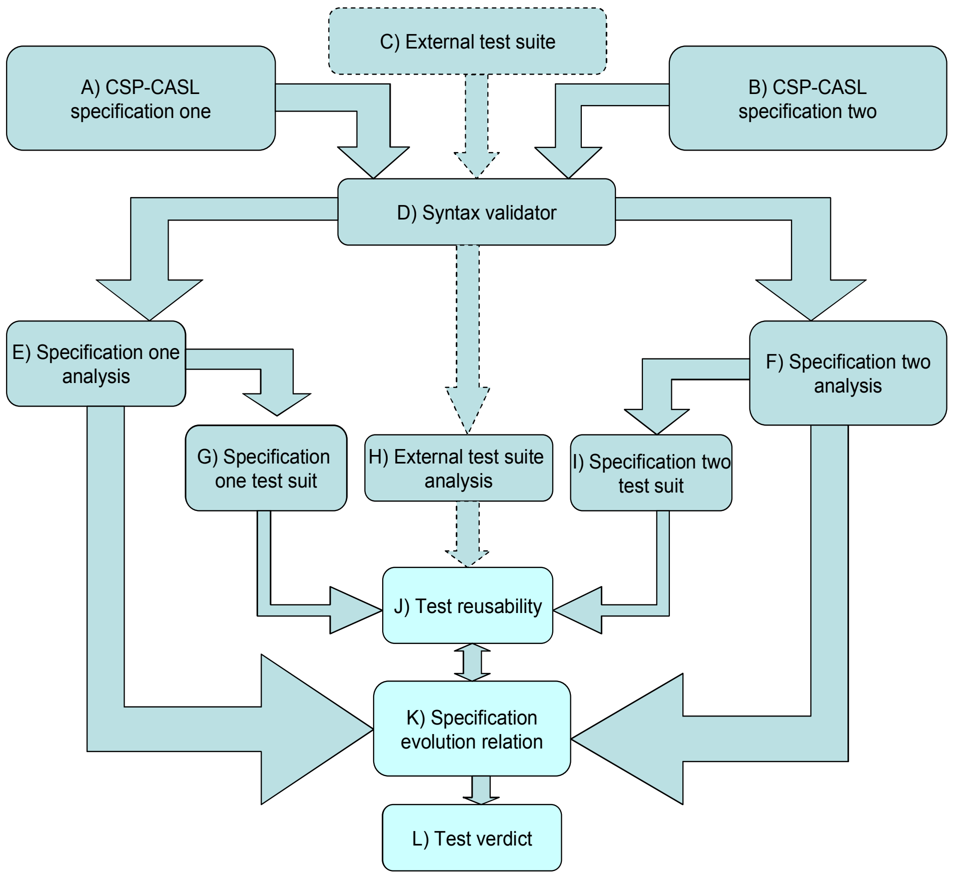

C) External test suite
A) CSP-CASL specification one
B) CSP-CASL specification two
D) Syntax validator
E) Specification one analysis
F) Specification two analysis
G) Specification one test suit
H) External test suite analysis
I) Specification two test suit
J) Test reusability
K) Specification evolution relation
L) Test verdict

FIGURE 4.5: Evolution and testing tool architecture

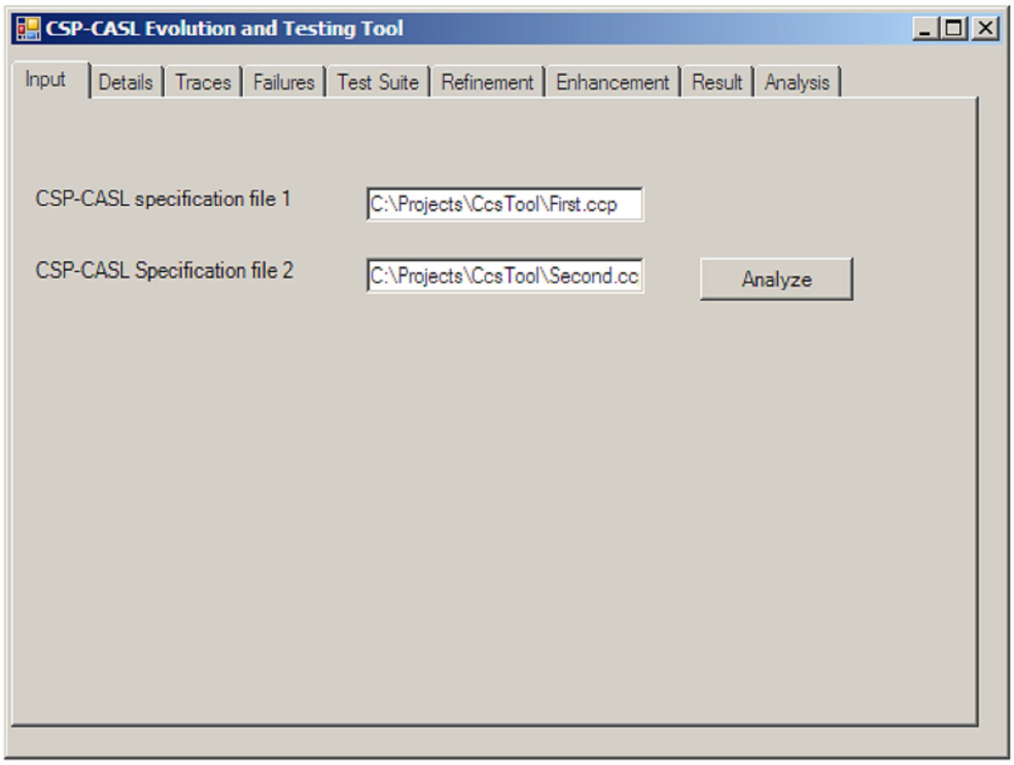
A and B. The analysis report is shown as data and process part details of the given specifications. Units G and I are test suite generation units based on the inputs from units E and F. Test suite generation is based on given definitions of positive and negative test generations. Units G,H,I also perform analysis on the test suites for further use. Unit K is a specification relation analysis unit which allows to present a relationship between given input specifications based on the definitions of software refinement and enhancement. Further, units J and L represent test suite reusability and test verdict analysis on the respective specifications. In the subsequent subsections, we will elaborate main features of *ccFormTest* by presenting respective screen shots.

4.5.2 Syntax validation

This part of the tool accepts CSP-CASL specifications and test suite as input and further passes them to the respective validations units. Syntax of input specification and test suite is validated according to the definitions as given in chapters

3 and 4. Syntax of any CSP-CASL specification is according to the specification formalism such as **ccSpec** *spName* = **Data** D_{obs} D_{int} [**Channel** Ch_{obs} Ch_{int}] **Process** P_{obs} P_{int} **End**. Data part and process part syntax are validated according to CSP-CASL syntax as given in the literatures [21] as well as in our proposed formalism. This tool implements only required CSP-CASL syntax validation possibility. Complete syntax validation is not considered as part of this tool development. This part of the tool is also used to validate syntax of test cases which are passed as an input.

A test case is defined as a positive or negative trace of CSP-CASL specification. Test case syntax is validated with CSP syntax over the signature of data part. Figure 4.6 presents the main screen of the *ccFormTest* tool. From this screen, the tool accepts CSP-CASL specifications as input which are syntactically validated once user presses the button *Analyze*. For the analysis of test suite, the tool accepts it as an input which can be provided from the another screen labeled as *Test Suite*. The test suite goes through syntax validation before it is ready for further analysis.



CSP-CASL specification file 1 C:\Projects\CcsTool\First.ccp

CSP-CASL Specification file 2 C:\Projects\CcsTool\Second.cc

FIGURE 4.6: ccFormTest, Main Screen

4.5.3 Specification evolution analysis

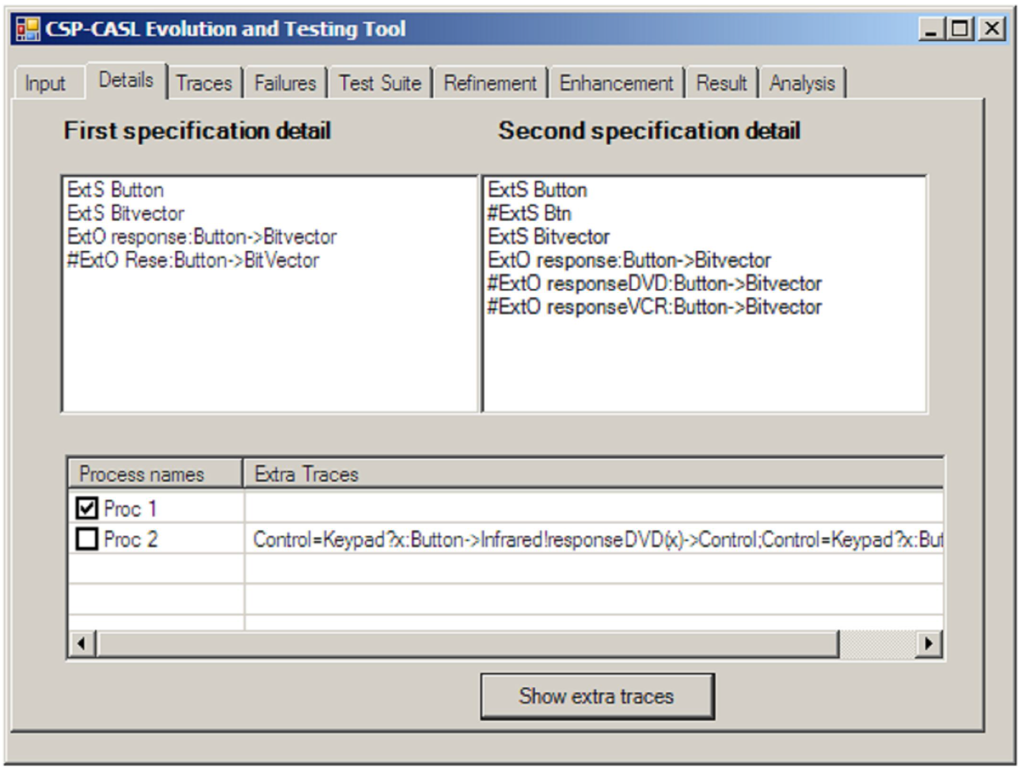
Specification evolution analysis is the most important functionality of this tool. This block of the tool accepts the specifications once it is passed through the syntax validation units. The CSP-CASL specifications are analyzed for establishing the relationship of specification refinement or specification enhancement. The specification refinement and specification enhancement analysis are carried out according to the definitions proposed in chapter three. In the analysis, observable and internal specification parts are considered for the specification relation analysis.

The result of analysis is presented in another screen labeled as $'Details'$. Here, the details of the data and the process part of specification are separately elaborated. Further, both the specification details are presented in a comparable view. This analysis unit acts as a basis for test generation and specification evolution relation. Specification evolution result is presented based on the definition of software enhancement and software refinement. Figure 4.7 presents the screen which shows the details of both the specifications. This functionality also allows to view the extra traces of selected specification, which is presented in the last part of the screen.

4.5.4 Test generation

Test generation is based on the simple definitions of test case as given in the previous part of this chapter. This tool generates both positive and negative test cases from the given specifications. A simple interface allows to generate test cases from selected input specifications. Any trace or failure of the input specification can be selected as a test case. At the tab of $Traces$ and $Failures$, all the traces and failures are presented from the selected input specification. These traces and failures can directly be used as a test case for the selected specification.

Figure 4.8 and 4.9 present the screen shots for the test case generation functionality. Further, it allows to select traces or failures as a test case from the testing of input specifications. The test verdict analysis is presented by selecting the test case and running it against the specifications into the screen 4.10. The external test suite can also be integrated into the tool once it is passed through syntax



First specification detail

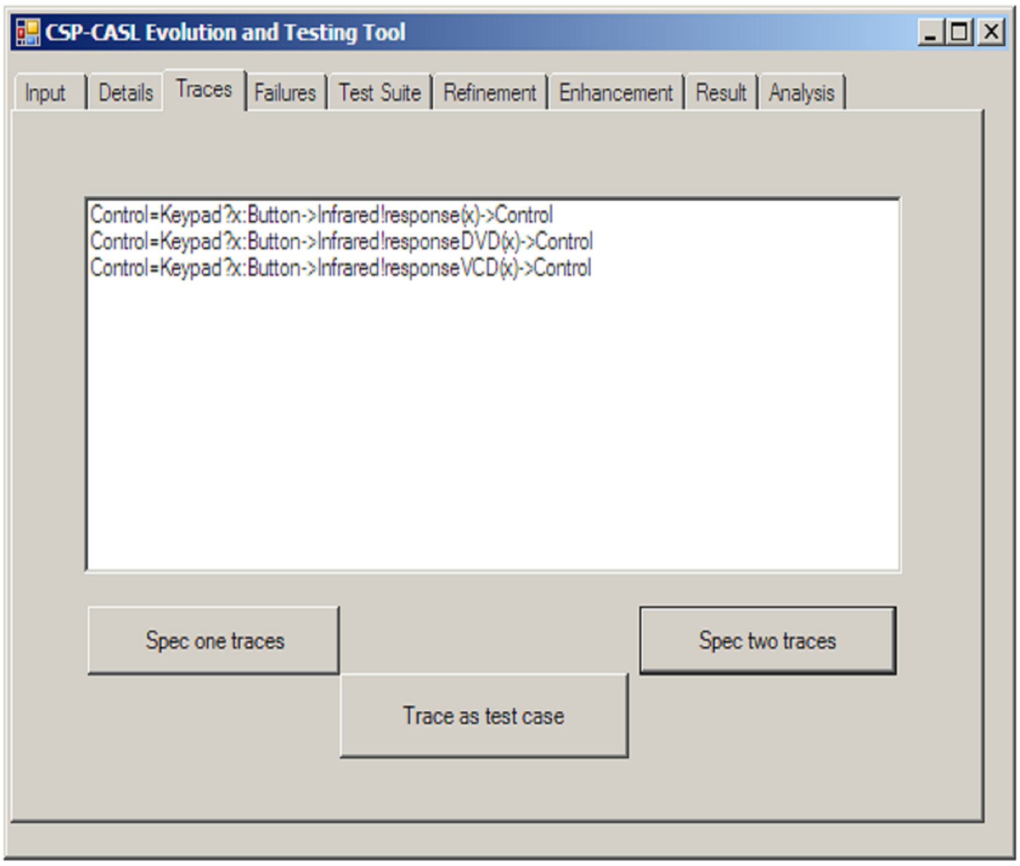
```
ExtS Button
ExtS Bitvector
ExtO response:Button->Bitvector
#ExtO Rese:Button->BitVector
```

Second specification detail

```
ExtS Button
#ExtS Btn
ExtS Bitvector
ExtO response:Button->Bitvector
#ExtO responseDVD:Button->Bitvector
#ExtO responseVCR:Button->Bitvector
```

Process names	Extra Traces
☑ Proc 1	
☐ Proc 2	Control=Keypad?x:Button->Infrared!responseDVD(x)->Control;Control=Keypad?x:But

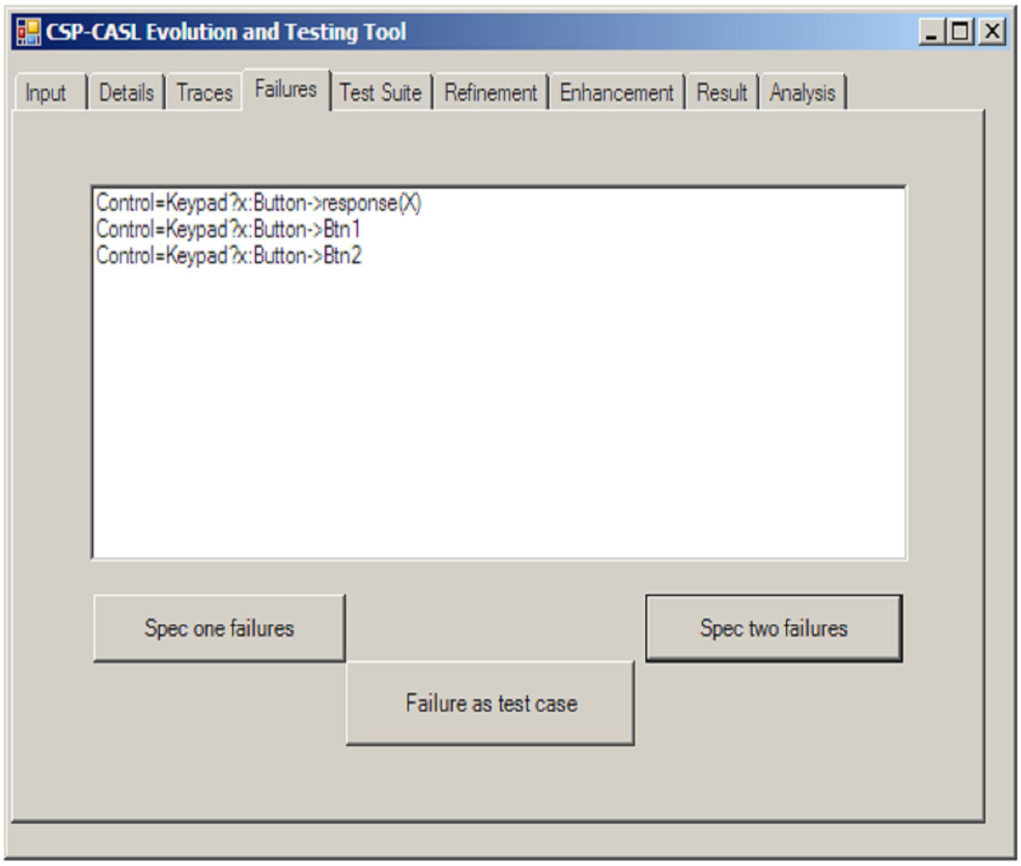
FIGURE 4.7: ccFormTest, Specification details



```
Control=Keypad?x:Button->Infrared!response(x)->Control
Control=Keypad?x:Button->Infrared!responseDVD(x)->Control
Control=Keypad?x:Button->Infrared!responseVCD(x)->Control
```

FIGURE 4.8: ccFormTest, Specification Trace and Positive Test Case

validation mechanism. External test cases can be analyzed for the testing of both the input specifications.



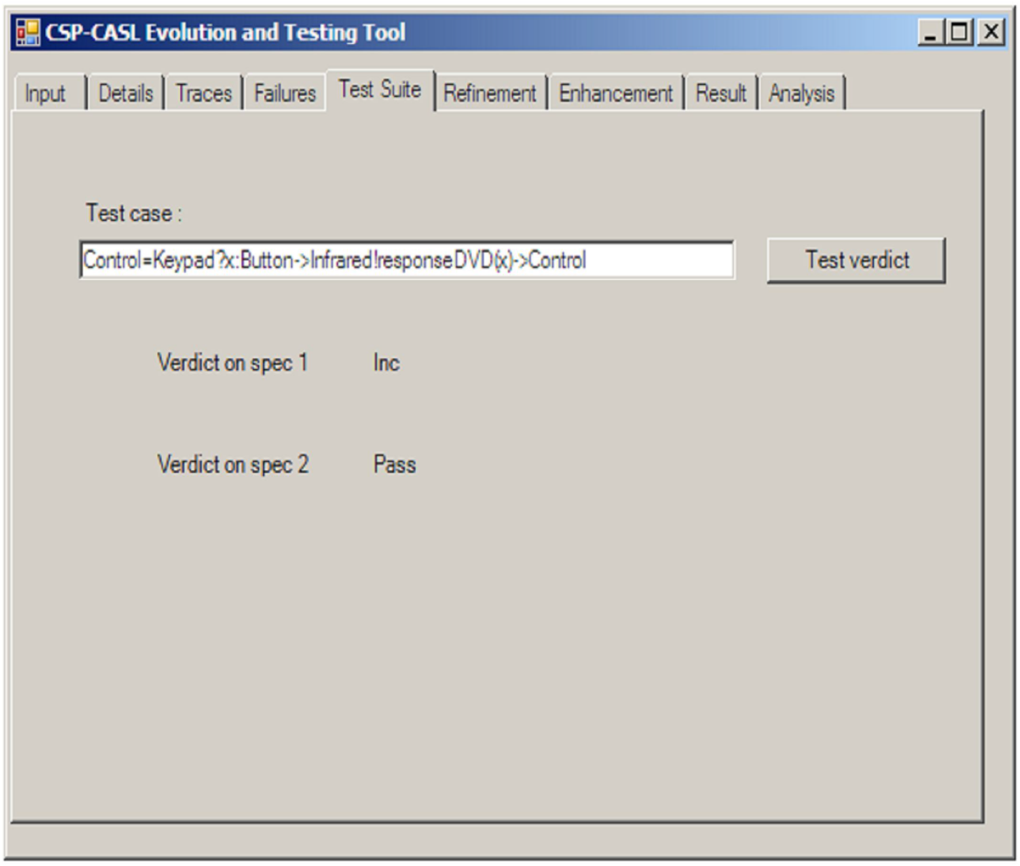
```
Control=Keypad?x:Button->response(X)
Control=Keypad?x:Button->Btn1
Control=Keypad?x:Button->Btn2
```

FIGURE 4.9: ccFormTest, Specification Failures and Negative Test Case

4.5.5 Test case evaluation and maintenance

This tool allows to analyze test cases on the given specifications. Especially, test verdicts on the given specification are decided by the definition of test verdict as given in previous part of this chapter. Once a test case is selected and button '*Trace as test case*' is pressed, then test verdicts from both the specifications are presented. This also allows to view the transition of the test verdict from one specification to another specification. The transition of the test verdict presents an initial view on the maintainability of the test case. Figure 4.10 presents a test case and its verdict on the selected specifications.

The test case reusability algorithm is implemented on the given definitions in the previous part of this chapter. The possibility of test cases are refinement and enhancement are considered at a very primitive level. The complete implementation



Test case :

Control=Keypad?x:Button->Infrared!responseDVD(x)->Control

Verdict on spec 1 Inc

Verdict on spec 2 Pass

FIGURE 4.10: ccFormTest, Test Case and its Verdict

of refinement relation requires interface with many other theorem provers which is out of scope for the thesis. However, CSP-CASL refinement relation has been nicely integrated into the tools developed by research group from Roggenbach [21]. The screen shot of testing related functionality is shown in the Figure 4.9. The test execution result is presented for both the input specifications. The tool gives the possibility to understand the transition of test verdict from one specification to another specification.

4.5.6 Specification and test case traceability analysis

ccFormTest allows to analyze a relation between two input specifications. Particularly, this relation is presented as specification refinement or specification enhancement. Once this relation is established, it allows to understand the properties of testing artifacts. This tool specially allows to trace a test case from one specification to another. Once a software refinement relation is established, it allows to prove the properties of test cases from one specification to another. Traceability is the basis to prove the CMMI process model properties which are described in chapter five.

This tool reads two input specifications; subsequently it provides relation between these two specifications. The specification relation is presented as specification refinement or specification enhancement. Further, this tool gives the possibility to derive test cases from input specifications. The test cases are further executed on both the provided specifications and a comparable view of test verdict is presented. The test case derived from initial specification can be executed on the enhanced specification and their test verdict is presented. Similarly, the test cases derived from abstract specification can be executed on the refined specification and their test verdict is presented. This allows to understand test case characteristics from abstract to refined and from initial to enhanced specification. This tool provides basic concept for specification and test suite traceability which is one of the requirements for the CMMI process compliance. The complete details of this are presented in the next chapter.

4.5.7 Tool summary

ccFormTest has assisted to prove our theoretical concepts of formal method based process and product quality improvement. This tool allows to analyze CSP-CASL based specification and elaborates a possibility of test suite reusability. Test suite reusability is especially elaborated in the software evolution. An approach of positive and negative test generation is elaborated based on our theoretical definitions.

This tool is developed for a limited set of CSP-CASL syntax and semantics. Integration of complete CSP-CASL syntax and semantics does not add much value to this research, only tool applicability will be enhanced. However, *ccFormTest* architecture supports interfaces with other tools, which will allow this to connect with other CSP-CASL tools for syntax validation and theorem provers. Integration with other tools can be considered as a further enhancement of this research, current state of art for this tool is limited with set of syntax and semantics.

Chapter 5

Process model compliance framework

The importance of formal methods for the evolution of software systems has been elaborated in the previous chapters. Generally, formal methods are used in the software system development life cycle. Initially, the software system development life cycle was confined to requirement, design, verification and validation activities. However, nowadays, the software system life cycle has evolved from the development level to the organization level. The software system life cycle does not end with the implementation, but it is extended to the maintenance and evolution phases of the developing software system. The contribution of formal methods at all phases of software system development is not well understood.

Generally, process improvement models are used to manage software systems at an organizational level. Some well known process improvement models are CMMI, ISO family, SPICE etc. Process improvement models and formal methods are viewed as two separate approaches for the development of software systems. However, they share the common goal to improve software system quality. In this research, we propose a unique approach to integrate these two aspects of software system quality improvements.

There are many process improvement models which have shown benefits to the organizations. Each of these process improvement models provides a collection of best practices for development and delivery of high-quality software systems. Process improvement model CMMI provides twofold guidelines for the development and maintenance of products. The first guideline is related to the process framework setup and the second guideline is related to the measurement of organizational capability and maturity for adopting the processes.

A software system development and maintenance life cycle is generally managed with two type of activities. First, *what* has to be developed as a software system and second, *how* the software system has to be developed and maintained. The formal methods have been explored right from the writing of specifications to the implementation of the requirements. The vast applicability of formal methods for the development and maintenance of the software system makes it suitable for the exploration of the *what* and *how* aspects of software development. In this research, we explore the compliance possibilities for process model CMMI(Capability Maturity Model Integration) with formal method based software development. The proposed compliance framework is elaborated with CSP-CASL based formalism methodologies.

5.1 CMMI, Capability Maturity Model Integration

A process model delivers process improvement results for software system development when it is rightly used in the right environment. A process improvement is concerned with changes, which can be implemented by a series of small steps to reach an improved state or it can be applied as a complete change. There have been many process improvement models supporting such types of industrial demands. These types of process improvement models are based on continuous process improvement (CPI) [2] or on the business process re-engineering (BPR) approach. In this research, we have selected continuous process improvement model CMMI . This process improvement model allows two types of implementations which are referred as *Maturity Levels* and *Capability Levels*. First, we describe architecture of CMMI in the next subsection which is required to explain these levels of CMMI implementation.

5.1.1 CMMI, process model architecture

CMMI (Capability Maturity Model Integration) is a framework for assisting organizations to improve their product development and maintenance process [33].

CMMI is based on the notion of *Process Area (PA)*. A *process area* is a cluster of related practices in an area. CMMI has 22 *process areas* which are considered important for the process improvement of an organization. CMMI offers two representations for its implementation, a continuous representation and a staged representation. The continuous representation offers more flexibility for process improvement. An organization can choose a focused process area, determine the dependent process areas, improve these at priority, and then concentrate on other process areas. In the staged representation, process areas are grouped together into capability maturity levels.

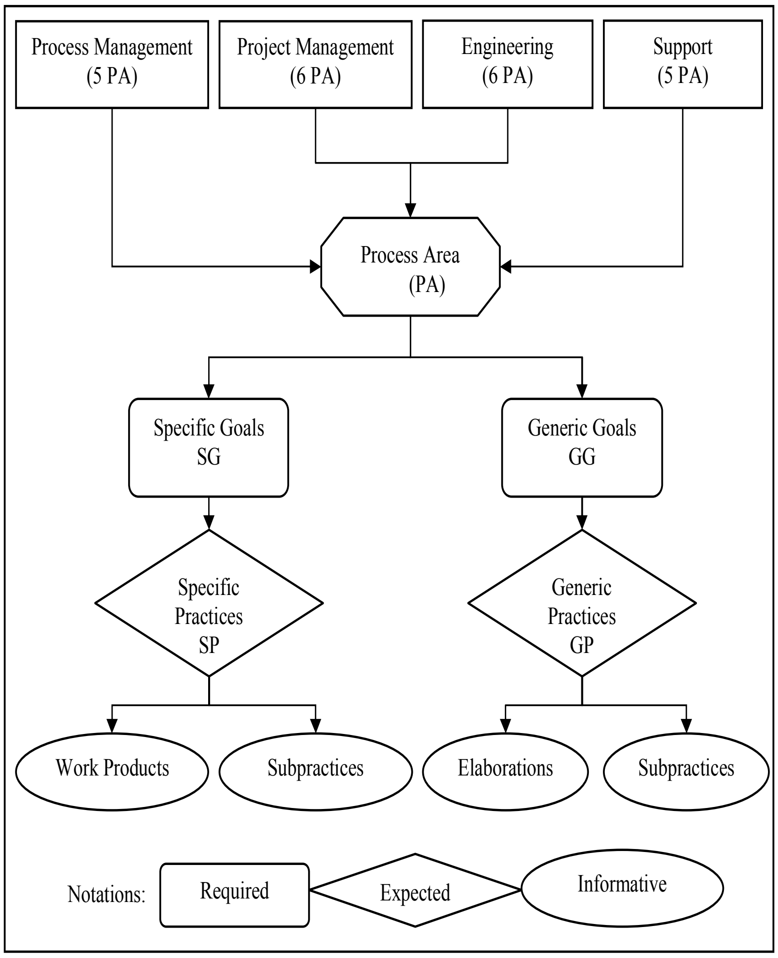


Figure 5.1: Details of process area and its components

Let us see how the CMMI architecture fits into the capabilities of an organization. In general, the domain of an organization can be divided into four groups: process management, project management, engineering and support (as shown in Figure 5.1). These groups have a set of business functions associated with them. Generally, these business functions have a quite independent set of business activities. Each organizational group has a set of process areas for improving the capabilities of its processes. Each process area is associated with a set of goals which have to be satisfied as a measure for the improvement in that process area. CMMI describes these aspects of a process area by so-called model components.

5.1.2 CMMI, process area model components

The CMMI process area is described by three types of components, they are referred to as model components. The compliance of CMMI process area is evaluated by compliance of its model components. These model components are referred to as follows:

- Required model components
- Expected model components
- Informative model components

Required model components describe what an organization must achieve to satisfy a *process area. Expected model components* describe what it may implement to achieve the associated required model components. *Informative model components* provide details which help to initiate the approach followed by required and expected model components (as shown in Figure 5.1). The description of a *process area* starts with an introduction, purpose and relation with other *process areas.* These are informative model components. The main characteristics of a *process area* are described by following two types of goals:

- Specific Goals(SG)
- Generic Goals(GG)

The specific goals are unique characteristics that must be present to satisfy the associated *process area.* A specific goal is a required model component. A *Specific Practice* (SP) is the description of an activity that is considered important in achieving the associated specific goal. A specific practice is an expected model component. A generic goal is the required characteristics component to institutionalize the processes which implement a *process area.* Generic goals are called *generic* because the same goal applies to multiple *process areas*. A generic practice is a description of an activity that is considered important in achieving the associated generic goal. Thus, a generic practice is an expected model component. For analyzing the compliance of CMMI process improvement models, we experiment with unique characters of the process areas e.g. specific goals and its specific practices. Generic goals are specially helpful to institutionalize a *process area* in the organization. The generic goal compliance is briefly investigated after specific goal compliance mechanism.

5.2 CMMI process improvement model and Formal methods

It is a well known fact that product quality depends on its development processes [78]. A good process is usually required to produce a good quality product. The development of a good quality software system on a predictable schedule and planned costs is possible only with an efficient process model. Some exceptions are also proved against the process model but these exceptions are proven for the development of small software systems. The broadening area of computer applications and interaction with different software systems is growing software system size day-by-day. Subsequently, an efficient process model becomes part of any software system development.

For the development of a software system, which nevertheless is reliable, both product and process based quality assurance methods are necessary. Process and product view of the quality assurance are the two main aspects for the development and maintenance of software systems. Several standard models have been proposed for a systematic process improvement, e.g., CMM/CMMI, Agile, SPICE, or the ISO 9000, Six-sigma family. For a rigorous analysis of the software products, formal method based software development has been proposed. Some examples of formal languages are VDM, Z, LOTOS, CSP and CASL etc. These two approaches are distinctly used for the development of software systems. However, the goal of these aspects is to achieve best quality in the developed software system. Some aspects of product and process quality views are brought together by research papers such as [1] [79][6]. However, an integration of these two aspects is still not understood in the theoretical and practical world of computer science. In our research, we integrate product and process quality aspect of software system with the formal methods based development approach. In particular, the presences of formal methods are extended for software system development as well as for the compliance of process improvement models.

5.2.1 Formal methods based idealistic approach to software development

Software development process is divided into several stages to efficiently build a software product within a specific budget and time frame. Generally, these stages are common to every software development approach such as Waterfall model, V-Model, Spiral Model, Iterative model, Agile development etc. Figure 5.2 shows an ideal software development approach which has all the software development stages. On the right hand side of these development stages we have collected the formal methods based techniques which are supporting that particular stage. However, in reality there is not a single formal specification language which is appropriate for all the stages of software development. This is where we call this as ideal software development, we assume there is a single formal specification language which fulfills the requirements for all stages. Once we collect the formal methods based techniques together, they look like as they are shown in the middle part of Figure 5.3. In our research, we refer them as formal method features. Later on, we investigate them for the compliance of process improvement model.

As shown in Figure 5.3, these features are common to any specification language and they are separately explored with most of the specification languages. Generally, the development of a large software system requires various formal specification languages for a precise description of required properties. Some integrated specification languages have been proposed for the development of such a software system. However, the proposed formal method features are fundamental properties for any specification language. Figure 5.3 presents the formal method features which are well established for the formal development of a software system. These features are further investigated for the compliance of selected process improvement model. In particular, we explore a formal specification language (CSP-CASL) based software development and investigate the properties of this language for the compliance of CMMI requirements.

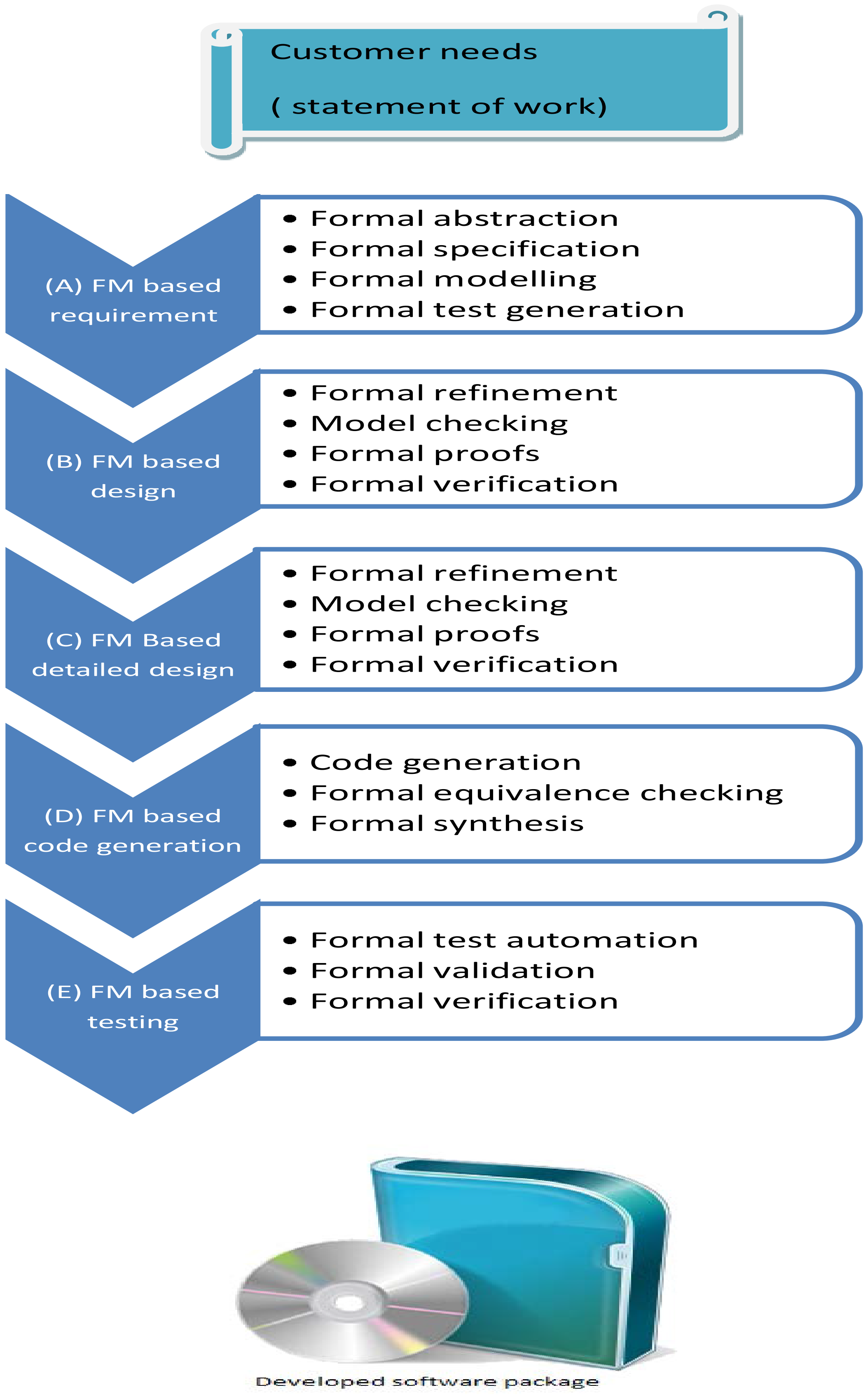


FIGURE 5.2: Formal method based idealistic approach to software development.

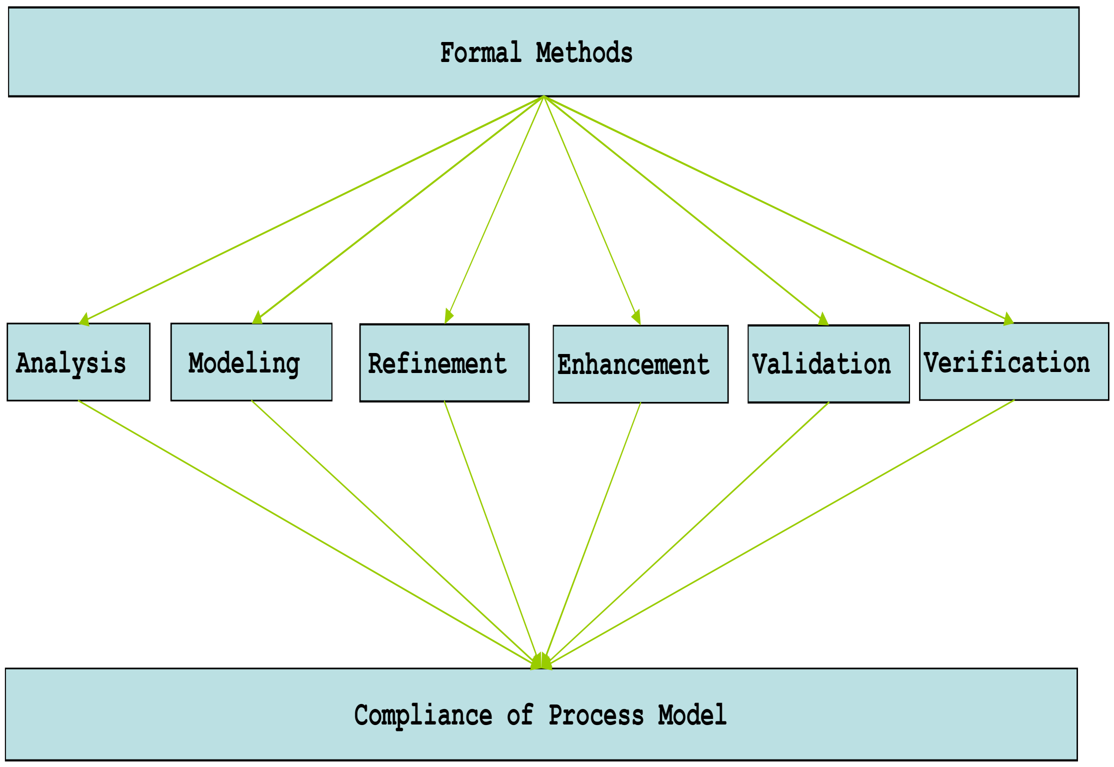


FIGURE 5.3: Formal method features appropriate for the compliance of CMMI process area.

This is the first step to systematically combine formal methods based techniques with process improvement models. The proposed idea is not only to bring formal methods and process improvement model together but to develop a distinct approach to the compliance of the process improvement model. The advantage of formal methods in software development life cycle automation brings a possibility to automate the compliance of the process improvement model. This research will elaborate an understanding of product and process quality parallel to the software development life cycle. In the next section, we give a brief overview of the process model compliance grading scheme for the compliance of CMMI components. In the subsequent sections our results about the contribution of formal methods in the compliance of CMMI process areas are presented.

5.2.2 Process model compliance grading scheme

We start our research activities of formal method based process model compliance by giving a definition of compliance grading scheme. This definition is used to evaluate compliance of the CMMI *process area* and its model components with formal method based software development. These grading schemes evaluate the compliance level of the model components of the selected process area with a formal method based development. The grading scheme proposes the following levels of evaluation:

- **Fully Complied (FC)**: A process area is FC if 90-100% of its specific goals are achieved as FC. A specific goal is considered as FC if 90-100% of its activities can be performed with a formal method based development.
- **Largely Complied (LC)**: A process area is LC if 60-89% of its specific goals are achieved as LC or FC. A specific goal is considered as LC if 60-89% of its activities can be performed with formal method based development.
- **Partially Complied (PC)**: A process area is PC if 30-59% of its specific goals are achieved as PC or LC. A specific goal is achieved as PC if 30-59% of its activities can be performed with formal method based development.
- **Not Complied (NC)**: A process area is NC if less than 29% of its specific goals are achieved as NC or PC. A specific goal is NC when only 29% or less of its activities can be performed with formal method based development.

This grading scheme is proposed with a detailed analysis of process improvement model and its required components. This grading scheme is specially proposed for a compliance of CMMI process model within an organization. This is not a replacement to the standard SCAMPI (Standard CMMI Appraisal Method for Process Improvement), which is designated to provide quality rating to the implementation of CMMI models. The proposed grading schemes support for the assessment of CMMI with selected SCAMPI. A detailed analysis of CMMI implementation process and formal method specification based development approach allow to investigate CMMI process compliance analysis in terms of the above grading schemes. Similar results can be achieved with any formal specification language based development approach. However, we present our result with CSP-CASL . This research activity is a starting point to achieve the process compliance parallel to the product development with formal methods.

5.2.3 CMMI, Process model compliance algorithm

Recall that the CMMI process model is based on the concepts of *process areas*. A *process area* is a collection of related practices in an area. These practices are considered important for the improvement of selected *process area*. This means the compliance of a *process area* is based on the compliance of its practices. Based on the elaboration of CMMI model components, we develop the following algorithm for the compliance of any process area. The algorithm *PA-Compliance* evaluates a level of compliance for a selected *process area*. This evaluation is categorized with the proposed grading scheme based on the practices performed with formal method based development for a select process area.

Table 5.1: Algorithm PA-Compliance

Steps	Activities
1.	Select a Process Area (PA)
2.	For each Specific Goal (SG) of the selected PA
3.	For each Specific Practices (SP) of the SG
4.	Evaluate compliance level for the activities of SP with formal method based software development and assign respective compliance level from grading scheme (FC,LC,PC,NO)
5.	Evaluate compliance level for SG based on the grading of SPs
6.	Evaluate a grading scale of the process area based on the grading of all SGs

The algorithm *PA-Compliance* starts with a consideration of the *Process Area*. This algorithm evaluates the compliance level of process area and its components with formal method based development. Compliance level is evaluated with the grading scheme as given in previous subsection. In further subsections, we present a set of CMMI process areas which complied with the formal method based software development approach.

A specific goal describes unique characteristics that must be present to satisfy the process area. The activities of the specific goal are described by specific practices. The specific goals are achieved by performing the activities of specific practices. Specific practice is the starting point to achieve the specific goal which subsequently satisfies the process area.

5.3 Formal method based CMMI implementation strategy

The implementation of process model CMMI is a continuous process. However, it starts with the development of Software Process Database (SPDB). A SPDB is a depository where CMMI reference process guidelines are maintained. These guidelines are a collection of reference practices which always evolve for an incorporation of best practices from the various learning aspects. A software system development in the CMMI environment starts with the selection of appropriate guidelines from the developed SPDB for the organization. Ownership of SPDB lies with a group generally referred to as the Software Engineering Process Group (SEPG). SEPG is responsible for the continuous improvement of SPDB practices with the help of feedback from SQA (Software Quality Assurance) and PM (Project Manager) groups. As the name suggests, the SQA group is responsible for maintaining product and process quality for software system development. The PM is responsible for the development of the software system according to the customer's requirements by following best practices derived from SPDB. This is a general view of an organization which is compliant with the CMMI process model.

Here, we develop a strategy for the compliance of the CMMI process model with the formal method based software system development. The compliance of CMMI components have been elaborated in the previous sections. Here, the compliance of the process model is considered at an organizational level. This strategy of formal method based CMMI process model compliance starts with the development of a Formal Software Process Database (FSPDB). This depository consists of CMMI practice guidelines and their compliance approach with formal method based development. Some examples of these practice guidelines are the formal description of customer requirement, derivation of design document and formal test cases generation etc. The overview of our proposed strategy for formal method based CMMI implementation is shown in Figure 5.4.

In Figure 5.4, complete CMMI compliance process is presented along with responsible groups. SEPG and PM are responsible for creation and selection of process guidelines for the development of a software system. The process guidelines are enhanced by feedback from the PM and the SQA. The proposed compliance framework is a quite similar approach used in the industry. The only difference is in the selection and evaluation of guidelines which requires knowledge of formal

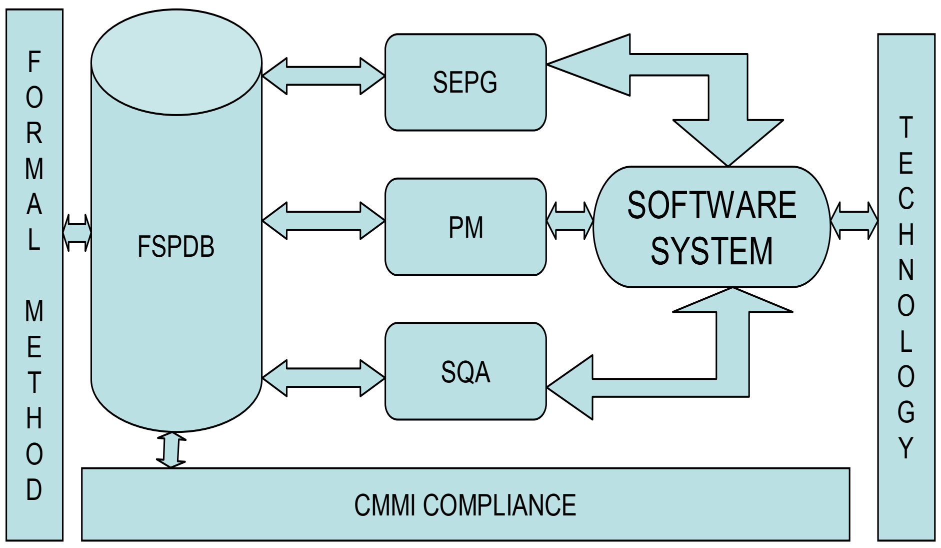


Figure 5.4: CMMI, process model evaluation strategy overview.

methods. The formal methods based SPDB provides various automation possibilities which subsequently reduces the involvement of SEPG/PM/SQA. The process model compliance evaluation is performed by the proposed algorithm and grading scheme. The CMMI compliance methodologies SCAMPI(A/B/C) [80] can be integrated with our grading scheme to evaluate process model implementation at an organizational level.

5.4 CMMI, Process area compliance exploration

The advantages of the formal method start with a precise and unambiguous description of the product requirements. Formalism in the product specification constitutes a basis for an automation possibility in the software development life cycle as well as in the software artifact traceability. Software artifact traceability is a process of tracking the product requirement and its components in other software life cycle phases. Formal method allows automatic traceability of software artifacts among software product artifacts e.g. requirement, design, detailed design etc.

In general, compliance of CMMI process model requires various tools such as requirement management tool, project management tool, quality management tool, time management tool, configuration management tool etc. Most of the organizations have to use different tools for performing these activities. There is a lack of tools which are appropriate for performing complete software development life cycle activities. One of the important reasons for this problem is the lack of standardization in the product requirement. Formal methods are by and large accepted as a standard way for writing and analyzing the specification. We extend the applicability of formal methods throughout the product development as well as for the process model compliance. The formal method based development approach has distinct properties where CMMI *process areas* can be satisfied parallel to the product development. Below are the list of process areas and their compliance grading scales based on our proposed process compliance algorithm *PA-Compliance*.

We explore the features of formal methods (Figure 5.3) for the compliance of CMMI process model. The compliance is evaluated with our proposed algorithm *PA-Compliance* and grading scheme with formal method based software system development. Formal methods based CMMI process area compliance is explored parallel to the product development. This means formal method based software system development gives a possibility to automate CMMI process area compliance with minimum extra effort. In further subsections, we present our theoretical evaluation of CMMI process areas which are complied with formal method based software development. We reached the conclusion that there are six process areas which can be satisfied up to a great extent with formal method based software development. Our results are achieved with CSP-CASL based formal software development however this approach is well suited for any specification language. In the next subsection, we present a list of process areas and their grading scales compliance with Formal Method Based Development(FMBD) by using Formal Method Features (FMF).

5.4.1 Requirements Management (RM)

The process area *Requirements Management* provides guidelines for addressing demands of product features and product component features. In addition to this, it also provides guidelines for removing inconsistencies between requirements and other work products. The compliance level of this process area and its component by formal method based software development is presented in Table 5.2. This table also presents the formal method features which are associated with each specific practices compliance.

Table 5.2: Compliance of RM with FMBD

Specific Goals and Specific Practices	**FMF**	**Grade**
SG 1 Manage Requirements	-	LC
SP 1.1 Obtain an Understanding of Requirements	Analysis, Modeling	LC
SP 1.2 Obtain Commitment to Requirements	-	NC
SP 1.3 Manage Requirements Changes	Enhancement	LC
SP 1.4 Maintain Bidirectional Traceability	Refinement, Enhancement	LC
SP 1.5 Identify Inconsistencies	Refinement, Enhancement	LC

This process area is specially related to the management of user requirements in such a way that completeness and consistency of requirements is maintained throughout the software product development. Formal specification based software development is significantly elaborated for writing user requirement, design document and test case generation. In the previous chapters, we presented a pragmatic definition of software specification, refinement, enhancement and test case, they all together establishes the basis for compliance of this process area. Let us investigate how specific goals of this process area are achieved by performing the specific practices with a formal method based development. Formal method based compliance evaluation of this process area is presented in the Table 5.2. In this table, is the analysis of results with our proposed algorithm *PA-Compliance*.

Here, the first step of the algorithm starts by selection of the *process area Requirements Management*. The next step is the selection of a *Specific Goal* from the selected PA. Here we select SG 1. The next step of the algorithm is to select a *Specific Practice* of the selected SG and assign evaluation with formal method based development. Here, first SP of selected SG is SP 1.1 which is *Obtain Understanding of Requirements*. Activities of this SP expect complete and clear understanding and management of user requirements. A formal method is well known for the implementation of these activities. The only concern with formal method based development remains with the training and the specification writing effort which bounds the number of users. With the consideration of these aspects for this SP our proposed grading scheme evaluates it at the level *LC*. The next step of the algorithm *PA-Compliance* is to select all SPs of selected SG. Depending on the formal specification generic features and evaluation of activities for each SP we assign SP 1.2 to SP 1.5 respective compliance level as shown in Table 5.2. Compliance grading of SG is the average of all its SPs grading. Here SG 1 has grading *LC* which is the average of the assigned grading to its SPs. Average of the SGs grading is the grading of the Process Area. Here, it's only one SG for this PA so the PA has a grading as *LC*.

A precise and unambiguous semantic of formal development is basis for the compliance of this process area. First, user requirement is formally specified and further this formal specification is formally extended in the software development life cycle. This approach of software development provides better software artifact traceability and management which forms a basis for the compliance of the specific goal and its specific practices for the Process Area *Requirements Management*.

Example 1. A small part of the previously proposed MED case study, is shown in the table 5.3 to demonstrate a transition of user requirement in the software development life cycle. The refinement relation among requirement, design and detailed design allows to trace the inconsistencies in the requirement. Our proposed tool *CcFormTest* allows us to verify the complete example formalism and its development life cycle with relations among refinement, enhancement and generated test cases.

Table 5.3: Refinement relation

In requirement	EncrMsg → SendMsg→ RecvMsg→ CheckAck→ TRUE
In design	FormatAck→ GenData →EncrMsg →SendMsg →RecvMsg→ CheckAck →TRUE \ { FormatAck , GenData } (Hiding internal functions makes equivalent to requirement)
In test case	EncrMsg→ SendMsg→ RecvMsg→ CheckAck →TRUE

5.4.2 Product Integration(PI)

The process area *Product Integration* guides the integration of the component's functions according to the requirements and the integration of components with a complete product. The contribution of formal method to this process area, specific goals and specific practices is shown in Table 5.4.

Formal method has been proposed for component based development, e.g. in [28]. In particular, CSP-CASL provides significant features for component based development, such as giving a structural and architectural approach to requirements engineering [67]. In addition to this, the advantage of CSP-CASL for product line

Table 5.4: Compliance of PI with FMBD

Specific Goals and Specific Practices	**FMF**	**Grade**
SG 1 Prepare for Product Integration	-	LC
SP 1.1 Determine Integration Sequence	Analysis, Modeling	LC
SP 1.2 Establish the Product Integration Environment	Analysis, Modeling	LC
SP 1.3 Establish Product Integration Procedures and Criteria	Analysis, Modeling	LC
SG 2 Ensure Interface Compatibility	-	LC
SP 2.1 Review Interface Completeness Descriptions	Analysis	LC
SP 2.2 Manage Interfaces	Modeling	LC
SG 3 Assemble Product Components and Deliver the Product	Analysis	PC
SP 3.1 Confirm Readiness of Product Components for Integration	Modeling	PC
SP 3.2 Assemble Product Components	Modeling	PC
SP 3.3 Evaluate Assembled Product Components	Verification, Validation	PC
SP 3.4 Package and Deliver the Product and Component	-	NC

based development has been studied in [24]. Process algebra [52] has very powerful features for mastering the complexity of processes via parallel and sequential composition. This process area compliance evaluation with formal methods based development is shown in Table 5.4.

5.4.3 Requirements Development(RD)

The purpose of this *process area* is to compile customer requirements, product requirements and product component requirements in such a way that it is clearly understandable by users. The process area component's compliance grading is presented in Table 5.5.

The formal method based unambiguous and precise description of product requirements are appropriate starting point for the compliance of SG 1 and SG 2. Compliance of SG 3 is achieved by formal method based validation and verification framework. Formal method based verification and validation have been established since decades and shown in various research articles [18] [42] etc. Briefly, we have described the basics of CSP-CASL based on our proposed validation framework in chapter three. The case study is evaluated in this validation framework for the compliance of this process area and the compliance result is presented in Table 5.5.

Table 5.5: Compliance of RD with FMBD

Specific Goals and Specific Practices	**FMF**	**Grade**
SG 1 Develop Customer Requirements	-	FC
SP 1.1 Elicit Needs	Analysis, Modeling	LC
SP 1.2 Develop the Customer Requirements	Modeling, Verification, Validation	FC
SG 2 Develop Product Requirements	-	FC
SP 2.1 Establish Product and Product Component Requirements	Analysis, Modeling, Refinement	FC
SP 2.2 Allocate Product Component Requirements	Modeling, Refinement	FC
SP 2.3 Identify Interface Requirements	Modeling	LC
SG 3 Analyze and Validate Requirements	-	LC
SP 3.1 Establish Operational Concepts and Scenarios	Analysis, Modeling	LC
SP 3.2 Establish a Definition of Functionality	Analysis	LC
SP 3.3 Analyze Requirements	Analysis	LC
SP 3.4 Analyze Requirements to Achieve Balance	-	PC
SP 3.5 Validate Requirements	Validation	FC

5.4.4 Technical Solutions(TS)

This *process area* provides guidance for design, development and implementation of the given requirements. The main focus of this *process area* is to evaluate and select a solution to develop a detailed design of the selected solution and to implement the design as a product or product component. Table 5.6 shows formal method based scale of compliance for this process area. The specification

Table 5.6: Compliance of TS with FMBD

Specific Goals (SG) and Specific Practices (SP)	**FSF**	**Grade**
SG 1 Select Product Component Solutions	-	LC
SP 1.1 Develop Alternative Solutions and Selection Criteria	Modeling, Refinement	LC
SP 1.2 Select Product Component Solutions	Modeling, Refinement	LC
SG 2 Develop the Design	-	PC
SP 2.1 Design the Product or Product Component	Modeling, Refinement	LC
SP 2.2 Establish a Technical Data Package	Refinement	PC
SP 2.3 Design Interfaces Using Criteria	Modeling	PC
SP 2.4 Perform Make, Buy, or Reuse Analyses	-	PC
SG 3 Implement the Product Design	-	LC
SP 3.1 Implement the Design	Refinement	LC
SP 3.2 Develop Product Support Documentation	Analysis, Modeling	PC

language based steps of refinement allow to establish traceability between abstract specification and design documents which subsequently leads to the possibility to generate an implementation code. In brief, the formal method based development [81] is well suited for the compliance of SG 1, SG 2, SG 3 and most of its specific practices. Below in Table 5.7, we show the aspect of refinement which is provable with our definitions given in chapter four.

Table 5.7: Refinement relation in SDLC elements

Requirement	**Design**	**Implementation**
Sort ComAck	ComAck = FormatAck(ComAck x SendData)	language based code

5.4.5 Validation

The purpose of the activities in this *process area* is to demonstrate that a product or product component fulfills its intended use when placed in its intended environment. The contribution of FMBD for this process is as follows in Table 5.8.

Table 5.8: Compliance of Validation with FMBD

Specific Goals and Specific Practices	**FMF**	**Grade**
SG 1 Prepare for Validation	-	FC
SP 1.1 Select Products for Validation	Analysis, Validation	LC
SP 1.2 Establish the Validation Environment	Analysis, Validation	FC
SP 1.3 Establish Validation Procedures	Modeling, Validation	FC
SG 2 Validate Product or Product Components	-	FC
SP 2.1 Perform Validation	Validation	FC
SP 2.2 Analyze Validation Results	Validation	LC

The formal methods based software development approach have major contributions to this process area. Starting from test case generation, test evaluation and test execution have been extensively experimented with formal methods based software development approach. They have been foundation for this process area compliance. We have developed a testing framework for CSP-CASL based test generation and execution which have already been elaborated into chapter three. In our consideration, each trace acts like a test case which has to be refined to be executable on the implementation. Steps of refinement should be similar refinement steps applied on specification. These are the basic considerations for our validation framework; this makes formal methods very appropriate for the compliance of SG 1 and SG 2.

5.4.6 Verification

The verification *process area* ensures that the products which are the result of the processes under improvement meet their specified requirements. The FMBD compliance grading of this process area is shown in the Table 5.9.

Table 5.9: Compliance of Verification with FMBD

Specific Goals and Specific Practices	**FSF**	**Grade**
SG 1 Prepare for Verification	-	LC
SP 1.1 Select Work Products for Verification	Verification, Validation	LC
SP 1.2 Establish the Verification Environment	Verification	LC
SP 1.3 Establish Verification Procedures	Verification	LC
SG 2 Perform Peer Reviews	-	NC
SP 2.1 Prepare for Peer Reviews	Refinement	PC
SP 2.2 Conduct Peer Reviews	Refinement	PC
SP 2.3 Analyze Peer Review Data	Refinement	PC
SG 3 Verify Selected Work Products	-	PC
SP 3.1 Perform Verification	Verification, Validation	LC
SP 3.2 Analyze Verification Results	Verification, validation	PC

Formal method based development has two ways to contribute to this process area, namely, model checking and theorem proving. Model checking is the process of building a model of a system and checking whether desired properties hold in the proposed model. Theorem proving is the process of finding the proof of a property from the axioms of a system, where the property and the system are expressed in the formal specification language [18]. An enormous amount of work has been done in these respects [42], [82]. Model checking and theorem proving have established significant presence in the industry especially in the development of complex systems. To investigate the compliance of this process area, formal method based techniques are evaluated with our proposed algorithm and results are presented in Table 5.9.

5.5 Compliance of Generic Goals (GG)

Compliance of the CMMI *process area* is not institutionalized until a process area has achieved its *generic goals*. As depicted in Figure 5.1 *generic goals* have associated *generic practices* which are expected model component. The expected model component explains the activities which are necessary to achieve CMMI model components. To achieve GG, their *generic practices* have to be implemented for the *process area* compliance at an organization level. CMMI degree of institutionalization is expressed with five levels of *generic goals* as shown in Table 5.10.

Formal method based CMMI *process area* compliance addresses the issues of institutionalization by its presence throughout the life cycle of product development.

TABLE 5.10: Compliance of Generic Goals with FMBD

Generic Goal	**Progression of Processes**	**Compliance with FSF**
GG 1	Performed process	GP 1.1 Perform Specific Practices
GG 2	Managed process	GP 2.1 Establish an Organizational Policy GP 2.2 Plan the Process GP 2.6 Manage Configurations GP 2.8 Monitor and Control the Process GP 2.9 Objectively Evaluate Adherence
GG 3	Defined process	GP 3.1 Establish a Defined Process GP 3.2 Collect Improvement Information
GG 4	Quantitatively managed process	-
GG 5	Optimizing process	-

Formal method based *process area* compliance contains one or more specific practices which fully implement *generic practices* that can be considered for the implementation of *generic practices*. Some of the *generic practices* which are implemented through the implementation of *specific practices* are shown in Table 5.10. *Generic practices* are common components to all *process areas*. The meanings of *generic practices* are interpreted according to the applying *process area*. Table 5.10 represents an overview of *generic practices* goals, further *process area* specific goals derived from these understandings.

The remaining *generic goal*s and their *generic practices* are more on the organizational issues. They are not in the context of formal method based product development. Table 5.10 presents the results based on the Specific Goal's Specific process mapping with *generic goal's generic practice*. The second column of the table presents the name of *generic goal* which has similar meaning as CMMI compliance maturity level description in earlier chapters.

5.6 CMMI representations and their compliance

Formal method based CMMI process model compliance is achieved via compliance of its *process area* and *specific goals* and *generic goals*. In the earlier sections, we have presented the *process areas* which are compliant with formal method based development. Process model CMMI is represented in the two ways in an organization; continuous representation and staged representation. Continuous representation uses the term capability level and staged representation uses the term maturity level. To reach a particular level, an organization must satisfy all the appropriate *process area* goals and set of targeted *process areas* based on the selected representation.

Our research is specially based on the formal methods based *process area* compliance. The *process areas* are common to both staged and continuous representation which makes our research of formal method based a compliance applicable to both representations. However, formal method based compliance is mostly related to the engineering related process areas which makes this proposal of process model compliance appropriate to continuous representation and to the capability level implementation in an organization. In continuous representation, selected process areas are implemented by achieving capability levels. Requirement gathering and engineering related process areas can be implemented with this approach by achieving various capability levels. Capability levels are means for incrementally improving the process corresponding to a given *process area.*

In the staged representation a set of *process areas* are grouped together to achieve organizational goal measured as maturity level. Each maturity level matures a uniquely defined set of *process areas.* Maturity levels are measured by achievement of *specific goals* and *generic goals* associated with set of *process areas.* Out of five maturity levels; formal method based development is suitable for the compliance of *process areas* of maturity levels one and two. Details of these *process areas* are given in the above subsections.

5.7 Summary

In this research of CMMI process model compliance with formal methods based development, we have reached to the significant contributions. First of all, this is a very distinct approach to the process model compliance where advantages from the process improvement model and formal method based software development are combined. This approach to process model compliance parallel to formal methods based software system development reduces process implementation effort and guarantee for a good quality product.

To investigate the CMMI process model compliance, parallel to product development, formal method based software development and maintenance approach is proposed. Formal method features are mapped with the prerequisites of CMMI process model. To establish a compliance level of the CMMI *process area* with formal method based development, a compliance grading scheme is proposed. This grading scheme is based on the achievement of *specific goals* of a *process area*. A generic algorithm is proposed for assigning a compliance level to a *process area*.

Table 5.11: CMMI process areas and their compliance level

Process area	**Formal method based compliance**
Requirement Management	LC
Product Integration	LC
Requirement Development	LC
Technical Solutions	LC
Verification	LC
Validation	LC

A part of an industrial case study is presented to illustrate the details of formal method based development for achieving the specific and generic goals of selected process areas of the CMMI process model. Out of 22 *process areas* from CMMI, six *process areas* can comply with a formal methods based product development. Our approach leads to the possibility of automation in process compliance which subsequently reduces the effort and cost for the implementation of a process model. In this research, we concentrate on CSP-CASL as a formal specification language however, our results are based on very generic features of specification formalism. Since a compliance result is achieved on very generic features of formal methods, it provides flexibility in the selection of any formal specification language. The similar compliance results can be achieved with any formal method based development approach.

Chapter 6

Summary and outlook

This is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.
Sir Winston Churchill.

Software product and process quality improvement has been the main goal of this thesis. This goal has been achieved on the foundations of formal methods and a process improvement model. In this chapter, we outline the results of our research and the prospective for future work. In the first part of this chapter the main result of our research work is highlighted. The current state of this research leads to various open questions which are discussed further in the second part of this chapter.

6.1 Results

The goal of this thesis has been to improve quality of software systems. This goal is achieved through approaching two aspects of software system quality: product quality and process quality. To achieve our research goal, we have proposed a novel framework for software product and process quality improvement. This framework is developed on the foundations of formal methods and a process improvement model. The complete framework is investigated with formal specification language

CSP-CASL and the process improvement model CMMI . The main outcomes of this research are described in the following paragraphs.

In chapter three, we proposed a distinct approach to software specification and software evolutions. We introduce a formal specification technique that allows to specify a software system in terms of observable and internal behaviors. The consideration of observable and internal behavior allows to elaborate a distinction between abstract and detailed specification in a pragmatic manner. The formalism of this specification approach is described by extending the syntax and semantics of the formal specification language CSP-CASL .

Further, vertical software evolution is proposed as a methodology of software refinement. The established ideas of software refinement are extended with the consideration of observable and internal behavior in the specification. In this research, the software refinement is proposed in a two step approach. In the first step, the existing refinement techniques are applied to the observable specification. In the second step, the internal specifications are refined by describing software design decisions. The internal specification refinement is referred to as constructive refinement. The complete syntax and semantics of this refinement approach is described with structural and behavioral specification language CSP-CASL .

In addition to vertical software evolution, horizontal software evolution is proposed as a methodology of software enhancement. The approach of software enhancement explores a methodology to manage software system changes and upgrades. The CSP-CASL based syntax and semantics are proposed for the enhancement of structural and behavioral properties of specifying system. The proposed formalism allows to investigate software enhancement and software artifact traceability.

Specially, CSP-CASL based software refinement and software enhancement is formally described and elaborated with an industrial case study. These formulations of software refinement and software enhancement play a key role to investigate the evolution of software systems. In the further research, these definitions are considered as a foundation for the development of product and process quality improvement framework.

In chapter four, we described a product quality framework. Particularly, we investigated testing methodologies for the CSP-CASL based specification formalism. Previously defined formalisms of software specification, software refinement and software enhancement are used to propose testing terminologies. Then the proposed testing terminologies are further elaborated in the vertical and horizontal software development paradigm.

The software systems are required to respond appropriately for expected as well as for unexpected behaviors. This requirement has extended a need of testing for expected behaviors as well as for unexpected behaviors. In this consideration, we extended the understanding of software testing with a direction of positive and negative test case generation. This research presents a distinct approach to test generation and test verdict interpretation during the evolution of software systems. Subsequently, given definitions are used to prove test artifact reusability theorems. The complete testing framework is elaborated with extended CSP-CASL based formalism.

The pragmatism of the proposed framework is supported with a development of a tool; *ccFormTest*. The main purpose of this tool is to elaborate test suite reusability during software evolution. This tool gives an abstract overview of test generation, test evaluation and test reusability. It is developed for a limited set of CSP-CASL syntax and semantics. Integration of complete CSP-CASL syntax and semantics does not add much value to this research, only tool applicability will be enhanced. However, *ccFormTest* architecture supports interfaces with other tools, which will allow this to connect with other CSP-CASL tools for syntax validation and theorem provers.

The chapter five describes the approach of the process quality improvement framework. The process quality improvement is considered as the compliance of process improvement model with an efficient approach. In this thesis, the process quality framework is proposed particularly for the compliance of the CMMI process improvement model. The compliance of the process model is based on the core aspects of the CMMI process model; the process areas. The process model compliance is proposed with formal methods based software development. Specially, the features of CSP-CASL are investigated for the compliance of the CMMI process model. The CMMI compliance grading scheme is developed to evaluate the level of compliance with formal method base software development. Further, a compliance algorithm is proposed to evaluate the process model through the evaluation of its components. The CMMI process areas are evaluated with the proposed algorithm. The result of compliance evaluation is presented in this thesis. The complete framework is supported with a developed tool which allows to practically support our theoretical concepts. Parallel to our theoretical contribution we work with an industrial partner where we applied the proposed framework for the development and maintenance of a medical instrument.

The understanding of the applicability of formal methods is extended to the organizational process model CMMI . Here, the complete framework is presented for formal specification language CSP-CASL and process model CMMI . However, a similar result can be achieved with other formal methods for the compliance with other process improvement models. This research is the starting point of process model compliance with formal methods. This has significant potential to automate the achievement of the process and product quality goals of software systems.

6.2 Future work

Formal methods and process improvement models have already been an important part of the software engineering domain. However, their presence is not a defacto standard for the development and maintenance of a software system. In this thesis, we have explored the possibilities of product and process improvement by investigating a relationship between formal methods and process improvement model. Our research has extended the presence of formal methods from product development/quality to process improvement. This research work is one of the foundational works for the product and process quality integration. This research still requires various scientific and industrial works to make this research results as a standard approach for any software industry. As a continuation of this research, we are proposing some directions of further work in the subsequent subsections.

6.2.1 Observable and internal specification formalism

A clear cut distinction between observable and internal specifications is not very common, whenever it comes to specify a software system. But this is practiced in almost all software projects. In this research, we have formulated this concept with a particular formal specification language CSP-CASL . Furthermore, it is required to enhance this concept into practice by investigating this approach with various specification mechanisms and programming languages.

6.2.2 Constructive approach to specification refinement

There have been various notions of refinement which have demonstrated significant advantages in certain scenarios. It is not advisable to develop a new notion of refinement for this specification formalism. Instead, other notions of refinement should be investigated to support this approach of specification. Particularly, a constructive refinement notion for existing approaches should be investigated to support addition of internal specification at the subsequent levels of descriptions.

6.2.3 Formal methods based positive and negative test case generation

Today's software industry is required to develop software systems which comply with requested features and behave properly under unexpected conditions. To develop such a software system, the requirements are generally written for the requested features as well as for the unexpected features. The fundamentals of these aspects have been considered for a long time but they are not well practiced. However, software development languages are well equipped to tackle such situations. In software development languages this is generally handled with exception handling methodologies. Very little research has been carried out for the verification and validation of such features. In this thesis, we considered an approach to generate test cases to test the requested as well as the unexpected features. However, we believe that more research with different types of software specification approaches is required.

6.2.4 Formal methods for process model compliance

In this thesis, formal specification language CSP-CASL has been investigated for the compliance of the CMMI process improvement model. The approach of process model compliance is independent of a particular formalism approach. However, to demonstrate this research we have used the CSP-CASL and the process compliance model CMMI . As an advancement to this research we would propose to investigate various combinations of process improvement models and formal methods within our proposed framework. In particular, we would suggest using model based formalism for the compliance of the process improvement model. This suggestion is based on industrial presence of model based development and process improvement models.

6.2.5 Industrial case studies and tool support

In this thesis, only a small part of an industrial application is considered as a case study to demonstrate applicability of the research results. However, to gain more confidence on the research results relatively large case studies should be developed within a CMMI certified organization.

Appendix A

List of publications

Some parts of this thesis are already published in various articles. The published articles are listed below:

- Test Case Reusability During the Evolution of Software System, to CS&P 2008
- Compliance of CMMI Process Area with Specification Based Development, SERA 2008, IEEE Conference
- Formal Specification Methods for the Improvement of Process and Product Quality, PhD paper FM08, Turku Finland
- Using Formal Specifications in the Implementation of CMMI, CS&P 2007, Poland
- CMMI Practices and Specification based development, SEE07, Munich Germany
- Specification Based Software Product Line Testing: A case study, CS&P 2006, Berlin Germany
- Towards Reusability of test suite during evolution of software systems, CALCO-Jnr UK Wales, 2005
- Loose semantics in the verification of communicating systems ETAPS-AVIS 2005 Edinburgh

Appendix B

List of articles which have cited this research

Some parts of this thesis have already been cited from various research articles. This research is cited by two different groups of scientific articles. Below is the list of articles which have referenced our process compliance work:

- Towards a framework to evaluate and improve the quality of implementation of CMMI practices, I Lopes Margarido, J Pascoal Faria - Product-Focused, 2012 - Springer
- Situational Process Improvement in Software Product Management WJ Bekkers - 2012 - igitur-archive.library.uu.nl
- Towards CMMI-compliant MDD Software Processes, AM Lins de Vasconcelos, G Giachetti - ICSEA 2011
- Utilizing VDM Models in Process Management Tool Development: an Industrial Case, CB Nielsen - Proceedings Of The 9th Overture - eng.au.dk, 2011
- CMMI / SPICE based process improvement, N. Ehsan, A. Perwaiz, J. Arif, E. Mirza, A. Ishaque,IEEE International Conference on Management of Innovation and Technology - IEEE ICMIT, 2010
- Sistema de gestao da certificaao de software: CMMI, ALF Lito - 2009 - ria.ua.pt

Following articles have referenced our software evolution and product quality improvement research:

- Strategies for Testing Products in Software Product Lines, I do Carmo Machado, JD McGregor, ES de Almeida, ACM SIGSOFT Software Engineering Notes, Volume 37 Issue 6, November 2012
- Regression Testing in Software Product Line Engineering, P Runeson, E Engström - Advances in Computers, 2012
- Software product line testing - A systematic mapping study, E Engström, P Runeson - Information and Software Technology, 2011 - Elsevier
- Variabilitätsmanagement in Anforderungs und Testfallspezifikation für Software-Produktlinien DWIA Wübbeke - 2010 - is.uni-paderborn.de
- Modeling variability and testability interaction in software product line engineering, M Jaring, RL Krikhaar, J Bosch - Composition-Based Software, 2008

Appendix C

Acknowledgments

This research has been carried out with direct and indirect support of many people. It is my pleasure to thank them for their contribution to make this thesis possible. The first person I would like to thank is my Supervisor Prof Dr Holger Schlingloff. I owe him lots of gratitude for providing support and encouragement, good teaching, critical suggestions, time and company that was always available, whenever I needed his advices and ideas. I would like to appreciate his teaching and guiding me for the writing of scientific articles and this thesis. Additionally, I would like to owe my thanks to thesis advisory committee members, Prof. Dr. Joachim Fischer and Prof. Dr. Bernd Krieg-Brückner and thesis committee chairman Prof. Dr. Niels Pinkwart for their time and effort to read my thesis and going through presentation.

Many thanks to Frau Heene for her ever smiling support and help in all respective administrative procedures and making life easier being in Humboldt- University. Also, many thanks to all members of the computer science department and library for their prompt and efficient support. Special thanks to Fraunhofer FIRST and its members where I have been given opportunity to have my desk and share ideas with our group.

I would thank Dr Roggenbach and his group for all scientific and non scientific discussions that is a part of my thesis. I am very much thankful to Mr Sumit Satpathy from IAL System GmbH, Dr Kallow from TeCNeT GmbH, Mr C. Rossenberg from Emersion Climate Technology GmbH, who provided me financial support as well as professional exposures with the understanding of scientific work. In addition to this I am thankful to Alfabet AG for providing me professional environment where

I am implementing my scientific exposure to practical world. Special thanks goes to my colleague and friend Arne Lex, Andrea Lex and Haider Karomi for their effort to translate my abstract into German language.

The support of Berlin friends who never made me feels homesick. Specially, Nagendra, Geeta and family, Ashok Uncle, Nisha Aunty and family, Abhay Bhaiya and Family, Bhumika and Ranjeet, Haider and Family, Chandresh and Family. The weekends and other social events were full of fun due to Aniket, Rashmi, Nimisha, Krishnan, Deepika, Rohan, Deepak, Shailaja, Ram, Umesh, Manu and Mahesh, Naveen and many more for providing some cherishable moments during my stay in Germany.

I am thankful to all my friends in India who kept me motivated though being distant apart. I wish to express my deepest appreciation to my parents: Geeta Mishra and Dayashankar Mishra, my late bade Papa Kripashankar Mishra, my brothers: Dr Ambrish, Rajkumar, Vijay and my sisters: Rekha, Meena and their families. Last but not the least I would like to thank my wife Shipra and her family for tolerating, encouraging and supporting me throughout my research.

Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig ohne unerlaubte Hilfe verfasst und nur die angegebene Literatur und die angegebenen Hilfsmittel verwendet zu haben.

Berlin, den

Bibliography

[1] Axel Van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications.* Wiley, March 2009. ISBN 0470012706. URL `http://www.worldcat.org/isbn/0470012706`.

[2] Paul E. McMahon. *Integrating CMMI and Agile Development: Case Studies and Proven Techniques for Faster Performance Improvement.* Addison Wesley, USA, 2010. ISBN 9780321714107. URL `http://www.sei.cmu.edu/cmmi/`.

[3] James R. Persse. *Process Improvement Essentials: CMMI, Six Sigma, and ISO 9001.* O Reilly Media, USA, 2006. ISBN 9780321713207.

[4] Michael L. George, John Maxey, David T. Rowlands, and Malcolm Upton. *The Lean Six Sigma Pocket Toolbook: A Quick Reference Guide to 70 Tools for Improving Quality and Speed.* Mcgraw-Hill Professional, USA, 2004. ISBN 0071441190.

[5] Zarina Shukur, Abdullah Zin, and Ainita Ban. *M2Z: A Tool for Translating a Natural Language Software Specification into Z*, volume 2495 of *Lecture Notes in Computer Science.* Springer Berlin / Heidelberg, 2002.

[6] Muan Ng and Michael Butler. Tool support for visualizing CSP in UML. *Formal Methods and Software Engineering*, pages 287–298, 2002. doi: 10.1007/3-540-36103-0_31. URL `http://dx.doi.org/10.1007/3-540-36103-0_31`.

[7] Joseph A. Goguen, James W. Thatcher, and Eric G. Wagner. *An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types.* Prentice-Hall, New Jersey, 1978.

[8] Paul P. Boca, Jonathan P. Bowen, and Jawedd I. Siddiqi, editors. *Formal Methods: State of the Art and New Directions.* Springer, 1 edition,

October 2009. ISBN 1848827350. URL `http://www.worldcat.org/isbn/1848827350`.

[9] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *Software Engineering, IEEE Transactions on*, 30(6):355–371, 2004. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1321059`.

[10] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison (Cambridge Tracts in Theoretical Computer Science).* Cambridge University Press, January 2001. ISBN 0521641705. URL `http://www.worldcat.org/isbn/0521641705`.

[11] W. W. Royce. *Managing the development of large software systems: concepts and techniques.* IEEE Computer Society Press Los Alamitos, CA, USA, Jan 1987. ISBN 0-89791-216-0.

[12] Klaus Pohl, Guenter Boeckle, and Frank van der Linden. *Software Product Line Engineering. Foundations, Principles, and Techniques*, volume 1. Springer, Berlin, USA, 2005. ISBN 3540243720.

[13] Katharine Whitehead. *Component-based Development. Principles and Planning for Business Systems.* Addison-Wesley Longman, Amsterdam, USA, 2002. ISBN 3540243720.

[14] Steve Reeves and David Streader. *Proceedings of the International Workshop on Formal Aspects of Computing.* Springer Verlag, 2007.

[15] Jim Mcall. *Software Quality Models and Philosophies*, volume 2. Encyclopedia of Software Engineering., USA, 1977. ISBN 0471377376.

[16] Sam Owre and Natarajan Shankar. A brief overview of pvs. *Theorem Proving in Higher Order Logics*, pages 22–27, 2008. doi: 10.1007/978-3-540-71067-7\_5. URL `http://dx.doi.org/10.1007/978-3-540-71067-7_5`.

[17] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Inc., October 1976. ISBN 013215871X. URL `http://www.worldcat.org/isbn/013215871X`.

[18] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, January 1999. ISBN 0262032708. URL `http://www.worldcat.org/isbn/0262032708`.

[19] Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. Guided test generation from CSP models. In *Proceedings of the 5th international colloquium on Theoretical Aspects of Computing*, pages 258–273, Berlin, Heidelberg, September 2008. Springer-Verlag. ISBN 978-3-540-85761-7. doi: 10.1007/978-3-540-85762-4_18. URL `http://dx.doi.org/10.1007/978-3-540-85762-4_18`.

[20] C. Nebut, F. Fleurey, Y. Le Traon, and J. Jezequel. Automatic test generation: A use case driven approach. *Software Engineering, IEEE Transactions on*, 32(3):140–155, 2006. doi: 10.1109/TSE.2006.22. URL `http://dx.doi.org/10.1109/TSE.2006.22`.

[21] Luc De Raedt, editor. *First Order Theory Refinement.* IOS Press, Amsterdam, 1996. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.2389`.

[22] Mary Beth Chrissis, Michael D. Konrad, and Sandra Shrum. *CMMI for Development: Guidelines for Process Integration and Product Improvement, Third Edition.* Addison-Wesley Professional, 1 edition, Jan 2010. ISBN 0-321-71150-5.

[23] Leo Freitas and Jim Woodcock. Fdr explorer. *Formal Aspects of Computing*, 21(1):133–154, February 2009. doi: 10.1007/s00165-008-0074-7. URL `http://dx.doi.org/10.1007/s00165-008-0074-7`.

[24] Alexandre Mota and Augusto Sampaio. Model-checking CSP-Z. *Fundamental Approaches to Software Engineering*, pages 205–220, 1998. doi: 10.1007/BFb0053592. URL `http://dx.doi.org/10.1007/BFb0053592`.

[25] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, December 1996. ISSN 0360-0300. doi: 10.1145/242223.242257. URL `http://dx.doi.org/10.1145/242223.242257`.

[26] J. A. Bergstra and J. W. Klop. Algebra of communicating processes. *Mathematics in Computer Science*, 1986.

[27] N. Rico, G. v. Bochmann, and O. Cherkaoui. Model-checking for real-time systems specified in LOTOS. In Gregor von Bochmann and David Probst,

editors, *Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 288–301. Springer Berlin / Heidelberg, 1993. ISBN 3-540-56496-9. URL `http://dx.doi.org/10.1007/3-540-56496-9-23`. 10.1007/3-540-56496-9-23.

[28] Peter D. Mosses and Michel Bidoit. *CASL User Manual: Introduction to Using the Common Algebraic Specification Language CASL.* Springer, January 2004. ISBN 354020766X. URL `http://www.worldcat.org/isbn/354020766X`.

[29] Peter D. Mosses, editor. *CASL Reference Manual: The Complete Documentation Of The Common Algebraic Specification Language*, volume 2960. Springer, March 2004. ISBN 3540213015. URL `http://www.worldcat.org/isbn/3540213015`.

[30] Egidio Astesiano. *Algebraic Foundations of Systems Specification (Ifip State-of-the-Art Reports).* Springer, 1999. ISBN 3540637729. URL `http://www.worldcat.org/isbn/3540637729`.

[31] Till Mossakowski, Anne Haxthausen, Donald Sannella, and Andrezj Tarlecki. CASL a common algebraic specification language. *Logics of Specification Languages*, pages 241–298, 2008. doi: 10.1007/978-3-540-74107-7_5. URL `http://dx.doi.org/10.1007/978-3-540-74107-7_5`.

[32] A. Roscoe. *Theory and Practice of Concurrency.* Prentice Hall, USA, 1997. ISBN 0136744095.

[33] Tony Hoare. Why ever CSP? *Electronic Notes in Theoretical Computer Science*, 162:209–215, September 2006. doi: 10.1016/j.entcs.2006.01.031. URL `http://dx.doi.org/10.1016/j.entcs.2006.01.031`.

[34] Watts Humphrey. *Managing the Software Process.* Addison-Wesley Professional (January 11, 1989), 1 edition, October 1989. ISBN 9780201180954.

[35] CMMI Product Team. CMMI for development, version 1.2, 1.3. *'CMU/SEI-2010-TR-033'*, 2006. URL `www.sei.cmu.edu/cmmi/`.

[36] D. Richard, Kuhn R. Chandramouli, and Ricky W. Butler. Cost effective use of formal methods in verification and validation. *Computer Security Division and CSRC*, 1982. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.970`.

[37] Donald Sannella. Algebraic specification and program development by stepwise refinement. *Logic-Based Program Synthesis and Transformation*, pages 1–9, 2000. doi: 10.1007/10720327_1. URL `http://dx.doi.org/10.1007/10720327_1`.

[38] Manfred Broy. Algebraic specification of reactive systems. *Theoretical Computer Science*, 239(1):3–40, 2000. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.4.9316`.

[39] Donald Sannella and Andrzej Tarlecki. *Algebraic Preliminaries*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. ISBN 3540637729. URL `http://portal.acm.org/citation.cfm?id=553529`.

[40] A. Hall. Seven myths of formal methods. *Software, IEEE*, 7(5):11–19, 1990. doi: 10.1109/52.57887. URL `http://dx.doi.org/10.1109/52.57887`.

[41] John Rushby. Theorem proving for verification. In *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 39–57. Springer Berlin / Heidelberg, 2001. doi: 10.1007/3-540-45510-8_2. URL `http://dx.doi.org/10.1007/3-540-45510-8_2`.

[42] Edmund Clarke. The birth of model checking. *25 Years of Model Checking*, pages 1–26, 2008. doi: 10.1007/978-3-540-69850-0_1. URL `http://dx.doi.org/10.1007/978-3-540-69850-0_1`.

[43] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971. ISSN 0001-0782. doi: 10.1145/362575.362577. URL `http://dx.doi.org/10.1145/362575.362577`.

[44] Joseph A. Goguen and Rod M. Burstall. Institutions: abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, January 1992. ISSN 0004-5411. doi: 10.1145/147508.147524. URL `http://dx.doi.org/10.1145/147508.147524`.

[45] Pierre Salverda, Grigore Roeyu, and Craig Zilles. Formally defining and verifying master/slave speculative parallelization. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 597–597. Springer Berlin / Heidelberg, 2005. URL `http://dx.doi.org/10.1007/11526841.10`. 10.1007/11526841-10.

[46] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, December 1972. doi: 10.1007/BF00289507. URL `http://dx.doi.org/10.1007/BF00289507`.

[47] Ralph J. Back and Joakim Wright. *Refinement Calculus: A Systematic Introduction (Texts in Computer Science)*. Springer, April 1998. ISBN 0387984178. URL `http://dx.doi.org/10.1007/11889229_99`.

[48] Ana Cavalcanti and Marie C. Gaudel. Testing for refinement in CSP. *Formal Methods and Software Engineering*, pages 151–170, 2007. doi: 10.1007/978-3-540-76650-6_10. URL `http://dx.doi.org/10.1007/978-3-540-76650-6_10`.

[49] Carroll Morgan. *Programming from specifications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990. ISBN 0137262256. URL `http://portal.acm.org/citation.cfm?id=95423`.

[50] Dirk Seifert. Conformance testing based on UML state machines. *Formal Methods and Software Engineering*, pages 45–65, 2008. doi: 10.1007/978-3-540-88194-0_6. URL `http://dx.doi.org/10.1007/978-3-540-88194-0_6`.

[51] Michael J. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27(2):139–173, September 1996. doi: 10.1016/0167-6423(96)81173-7. URL `http://dx.doi.org/10.1016/0167-6423(96)81173-7`.

[52] C. Hoare. Prespecification in data refinement. *Information Processing Letters*, 25(2):71–76, May 1987. ISSN 00200190. doi: 10.1016/0020-0190(87)90224-9. URL `http://dx.doi.org/10.1016/0020-0190(87)90224-9`.

[53] Ursula Goltz, Roberto Gorrieri, and Arend Rensink. On syntactic and semantic action refinement. In Masami Hagiya and John Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 385–404. Springer Berlin / Heidelberg, 1994. URL `http://dx.doi.org/10.1007/3-540-57887-0-106`. 10.1007/3-540-57887-0-106.

[54] Mark-Oliver Stehr, Jose Meseguer, and Peter Alveczky. Rewriting logic as a unifying framework for petri nets. In Hartmut Ehrig, Julia Padberg, Gabriel

Juhs, and Grzegorz Rozenberg, editors, *Unifying Petri Nets*, volume 2128 of *Lecture Notes in Computer Science*, pages 250–303. Springer Berlin / Heidelberg, 2001.

[55] R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. ISBN 0-13-115007-3. URL `http://portal.acm.org/citation.cfm?id=63446`.

[56] John V. Guttag. *The specification and application to programming of abstract data types.* PhD thesis, University of Toronto, Toronto, Ont., Canada, Canada, 1975. URL `http://portal.acm.org/citation.cfm?id=908660`.

[57] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* Springer-Verlag, 1992. ISBN 0387137181. URL `http://www.worldcat.org/isbn/0387137181`.

[58] Berthold Hoffmann and Bernd Krieg-Brückner. The prospectra system. In Christian Choffrut and Matthias Jantzen, editors, *STACS 91*, volume 480 of *Lecture Notes in Computer Science*, pages 539–540. Springer Berlin / Heidelberg, 1991. URL `http://dx.doi.org/10.1007/BFb0020829`. 10.1007/BFb0020829.

[59] Till Mossakowski, Donald Sannella, and Andrzej Tarlecki. A simple refinement language for CASL. In JosLuiz Fiadeiro, PeterD. Mosses, and Fernando Orejas, editors, *Recent Trends in Algebraic Development Techniques*, volume 3423 of *Lecture Notes in Computer Science*, pages 162–185. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-25327-3. doi: 10.1007/978-3-540-31959-7_10. URL `http://dx.doi.org/10.1007/978-3-540-31959-7_10`.

[60] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68, 1980.

[61] H. Ehrig, B. Mahr, I. Classen, and F. Orejas. Introduction to algebraic specification part i and part ii. *The Computer Journal 35 (5)(1993), 460 - 467*, 1993.

[62] Glenford J. Myers. *The Art of Software Testing, Second Edition.* Wiley, 2 edition, June 2004. ISBN 0471469122. URL `http://www.worldcat.org/isbn/0471469122`.

[63] Jan Tretmans. Testing concurrent systems: A formal approach. *CONCUR'99 Concurrency Theory*, page 779, 1999. doi: 10.1007/3-540-48320-9_6. URL `http://dx.doi.org/10.1007/3-540-48320-9_6`.

[64] B. Korel. Automated software test data generation. *Transactions on Software Engineering*, 16(8):870–879, 1990. doi: 10.1109/32.57624. URL `http://dx.doi.org/10.1109/32.57624`.

[65] Antoni Diller. *Z.: An Introduction to Formal Methods*. John Wiley & Sons, second edition, May 1994. ISBN 0471939730. URL `http://www.worldcat.org/isbn/0471939730`.

[66] Anja Ebersbach, Markus Glaser, and Richard Heigl. *Wiki : Web Collaboration*. Springer, November 2005. ISBN 3540259953. URL `http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/3540259953`.

[67] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, July 2003. ISBN 1930110936. URL `http://www.worldcat.org/isbn/1930110936`.

[68] M. Jackson. *Software Requirements And Specifications (Acm Press Books)*. Addison-Wesley Professional, August 1995. ISBN 0201877120. URL `http://www.worldcat.org/isbn/0201877120`.

[69] Michel Bidoit, Don Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. In *Proc. 7th Int. Conf. Algebraic Methodology and Software Technology (AMAST'98), Amazonia, Brazil, Jan. 1999*, volume 1548, pages 341–357. Springer, 1999. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.735`.

[70] Monroe Newborn and Monty Newborn. *Automated Theorem Proving: Theory and Practice*. Springer, December 2000. ISBN 0387950753. URL `http://www.worldcat.org/isbn/0387950753`.

[71] Temesghen Kahsai, Markus Roggenbach, and Holger Schlingloff. Specification-based testing for software product lines. *SEFM 2008 - Proc 6th IEEE International Conference on. Software Engineering and Formal Methods*, 2008.

[72] Jan Tretmans. Model based testing with labelled transition systems. *Formal Methods and Testing*, pages 1–38, 2008. doi: 10.1007/978-3-540-78917-8_1. URL `http://dx.doi.org/10.1007/978-3-540-78917-8_1`.

[73] Lars Frantzen, Jan Tretmans, and Tim A. Willemse. Test generation based on symbolic specifications. *Formal Approaches to Software Testing*, pages 1–15, 2005. URL `http://www.springerlink.com/content/uu3va76k39megkke`.

[74] Jon Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linköping*, pages 21–28. ECSEL, October 1999. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.963`.

[75] Hans-Martin Hörcher and Jan Peleska. Using formal specifications to support software testing. *Software Quality Journal*, 4(4):309–327, December 1995. doi: 10.1007/BF00402650. URL `http://dx.doi.org/10.1007/BF00402650`.

[76] Debra J. Richardson, Owen T. O'Malley, and C. Tittle. Approaches to specification-based testing. In *Symposium on Testing, Analysis, and Verification*, pages 86–96, 1989. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.2447`.

[77] Roy P. Pargas, Mary J. Harrold, and Robert R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999. doi: 10.1002/(SICI)1099-1689(199912)9:4\%3C263::AID-STVR190\%3E3.0.CO;2-Y. URL `http://dx.doi.org/10.1002/(SICI)1099-1689(199912)9:4%3C263::AID-STVR190%3E3.0.CO;2-Y`.

[78] Mary Beth Chrissis, Mike Konrad, and Sandy Shrum. *CMMI for Development: Guidelines for Process Integration and Product Improvement*, volume 3. Addison Wesley, 2008. ISBN 0321711505.

[79] Manfred Broy and Oscar Slotosch. Enriching the software development process by formal methods. *Applied Formal Methods â FM-Trends 98*, pages 44–61, 1999. doi: 10.1007/3-540-48257-1_2. URL `http://dx.doi.org/10.1007/3-540-48257-1_2`.

[80] Dennis M. Ahern, Jim Armstrong, Aaron Clouse, Jack R. Ferguson, Will Hayes, and Kenneth Nidiffer. *CMMI Scampi Distilled: Appraisals for Process*

Improvement. Addison-Wesley Professional (January 11, 1989), 1 edition, October 2005. ISBN 0321228766.

[81] Jifeng He and Tony Hoare. CSP is a retract of CCS. *Unifying Theories of Programming*, pages 38–62, 2006. doi: 10.1007/11768173_3. URL `http://dx.doi.org/10.1007/11768173_3`.

[82] Joël Ouaknine and Steve Schneider. Timed CSP: A retrospective. *Electronic Notes in Theoretical Computer Science*, 162:273–276, September 2006. ISSN 15710661. doi: 10.1016/j.entcs.2005.12.093. URL `http://dx.doi.org/10.1016/j.entcs.2005.12.093`.

[83] C. A. R. Hoare, Jayadev Misra, Gary T. Leavens, and Natarajan Shankar. The verified software initiative: A manifesto. *ACM Comput. Surv.*, 41(4): 1–8, 2009. ISSN 0360-0300. doi: 10.1145/1592434.1592439. URL `http://dx.doi.org/10.1145/1592434.1592439`.

[84] Jim Davies. *Using CSP*. LNCS, 2006. doi: 10.1007/11889229_3. URL `http://dx.doi.org/10.1007/11889229_3`.

[85] Till Mossakowski. CASL: From semantics to tools. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 93–108, 2000. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.9631`.

[86] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, volume 29, pages 97–107, New York, NY, USA, July 2004. ACM Press. ISBN 1581138202. doi: 10.1145/1007512.1007526. URL `http://dx.doi.org/10.1145/1007512.1007526`.

[87] K. Bogdanov and M. Holcombe. Refinement in statechart testing. *Software Testing, Verification and Reliability*, 14(3):189–211, 2004. ISSN 1099-1689. doi: 10.1002/stvr.301. URL `http://dx.doi.org/10.1002/stvr.301`.

[88] Bertrand Meyer. Seven principles of software testing. *Computer*, 41(8):99–101, 2008. doi: 10.1109/MC.2008.306. URL `http://dx.doi.org/10.1109/MC.2008.306`.

[89] Luciano Baresi and Mauro Pezzè. An introduction to software testing. *Electronic Notes in Theoretical Computer Science*, 148(1):89–111, February 2006. doi: 10.1016/j.entcs.2005.12.014. URL `http://dx.doi.org/10.1016/j.entcs.2005.12.014`.

[90] Antti Kervinen, Mika Maunumaa, Tula Päkkönen, and Mika Katara. Model-based testing through a gui. *Formal Approaches to Software Testing*, pages 16–31, 2006. doi: 10.1007/11759744_2. URL `http://dx.doi.org/10.1007/11759744_2`.

[91] I. S. W. B. Prasetya, T. E. J. Vos, and A. Baars. Trace-based reflexive testing of oo programs with t2. *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 151–160, 2008. doi: 10.1109/ICST.2008.12. URL `http://dx.doi.org/10.1109/ICST.2008.12`.

[92] David S. Dummit and Richard M. Foote. *Abstract Algebra*. Wiley, 3 edition, July 2003. ISBN 0471433349. URL `http://www.worldcat.org/isbn/0471433349`.

[93] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, July 1995. doi: 10.1109/52.391826. URL `http://dx.doi.org/10.1109/52.391826`.

[94] P. Alencar, D. Cowan, and C. Lucena. A formal approach to architectural design patterns. *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 576–594, 1996. doi: 10.1007/3-540-60973-3_108. URL `http://dx.doi.org/10.1007/3-540-60973-3_108`.

[95] Edmund M. Clarke, Jeannette M. Wing, Rajeev Alur, Rance Cleaveland, David Dill, Allen Emerson, Stephen Garland, Steven German, John Guttag, Anthony Hall, Thomas Henzinger, Gerard Holzmann, Cliff Jones, Robert Kurshan, Nancy Leveson, Kenneth Mcmillan, J. Moore, Doron Peled, Amir Pnueli, John Rushby, Natarajan Shankar, Joseph Sifakis, Prasad Sistla, Bernhard Steffen, Pierre Wolper, Jim Woodcock, and Pamela Zave. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.625`.

[96] David Garlan. Formal modeling and analysis of software architecture: Components, connectors, and events. *Formal Methods for Software Architectures*, pages 1–24, 2003. URL `http://www.springerlink.com/content/2xedpxdgg209gdb7`.

[97] Lijun Shan and Hong Zhu. A formal descriptive semantics of UML. *Formal Methods and Software Engineering*, pages 375–396, 2008. doi: 10.1007/978-3-540-88194-0_23. URL `http://dx.doi.org/10.1007/978-3-540-88194-0_23`.

[98] Jean F. Monin. *Understanding Formal Methods.* Formal Approaches to Computing and Information Technology. Springer, January 2003. ISBN 1852332476. URL `http://www.worldcat.org/isbn/1852332476`.

[99] Richard F. Paige and Phillip J. Brooke. Agile formal method engineering. *Integrated Formal Methods*, pages 109–128, 2005. doi: 10.1007/11589976_8. URL `http://dx.doi.org/10.1007/11589976_8`.

[100] Sten Agerholm and Peter Larsen. A lightweight approach to formal methods. In *Applied Formal Methods FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 168–183. Springer Berlin / Heidelberg, 1999. doi: 10.1007/3-540-48257-1_10. URL `http://dx.doi.org/10.1007/3-540-48257-1_10`.

[101] Ben Potter, David Till, and Jane Sinclair. *An Introduction to Formal Specification and Z.* Prentice Hall International Series in Computer Science. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1996. ISBN 0132422077. URL `http://portal.acm.org/citation.cfm?id=547639`.

[102] H. Ehrig, Groe M. Rhode, and U. Wolter. The role of category theory in the area of algebraic specifications, 1996. URL `http://citeseer.ist.psu.edu/ehrig96role.html`.

[103] Robin Milner. An algebraic definition of simulation between programs. Technical report, Stanford University, Stanford, CA, USA, 1971. URL `http://portal.acm.org/citation.cfm?id=891902`.

[104] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1-2):83–133, 1984. doi: 10.1016/0304-3975(84)90113-0. URL `http://dx.doi.org/10.1016/0304-3975(84)90113-0`.

[105] Iain Phillips. Refusal testing. *Automata, Languages and Programming*, pages 304–313, 1986. doi: 10.1007/3-540-16761-7_80. URL `http://dx.doi.org/10.1007/3-540-16761-7_80`.

[106] Michael Von Der Beeck. *Behaviour Specifications:Equivalence And Refinement*. CiteSeer, 2000. URL `http://citeseer.ist.psu.edu/673370.html`.

[107] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL: the Common Algebraic Specification Language. *Theoretical Computer Science*, 286 (2):153–196, September 2002. doi: 10.1016/S0304-3975(01)00368-1. URL `http://dx.doi.org/10.1016/S0304-3975(01)00368-1`.

[108] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. *Refinement: An overview*. LNCS, 2006. doi: 10.1007/11889229_1. URL `http://dx.doi.org/10.1007/11889229_1`.

[109] Christie Bolton and Jim Davies. Refinement in Object-Z and CSP. *Integrated Formal Methods*, pages 225–244, 2002. doi: 10.1007/3-540-47884-1_13. URL `http://dx.doi.org/10.1007/3-540-47884-1_13`.

[110] Satish Mishra and Bernd H. Schlingloff. Compliance of CMMI process area with specification based development. *SERA IEEE Computer Society*, pages 77–84, 2008. doi: 10.1109/SERA.2008.36. URL `http://dx.doi.org/10.1109/SERA.2008.36`.