Identifying the Suitability of Any Formal Language for a Particular Application Area

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Software Engineering

By: **Monika Dhariwal**
(80731014)

Under the supervision of:
Ms. Shivani Goel
Lecturer

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

MAY 2009

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Identifying the Suitability of Any Formal Language in a Particular Application Area"**, in partial fulfilment of the requirements for the award of degree of Master of Engineering in Software Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Ms. Shivani Goel* and refers other researcher's works which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

Monika.

(Monika Dhariwal)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Shivani Goel

(Ms. Shivani Goel)
Lecturer
Computer Science & Engineering Department
Thapar University
Patiala.

Countersigned by:

03/07/2009

(HEAD)
Computer Science & Engineering Deptt.,
Thapar University,
Patiala.

9/7/2009

(Dr. R. K. Sharma)
Dean (Academic Affairs)
Thapar University,
Patiala.

Acknowledgement

No volume of words is enough to express my gratitude towards my guides, **Ms. Shivani Goel**, Lecturer, Computer Science and Engineering Department, Thapar University, who have been very concerned and have aided for all the material essential for the preparation of this thesis report. She has helped me to explore this vast topic in an organized manner and provided me with all the ideas on how to work towards a research-oriented venture.

I am also thankful to **Dr. Rajesh Bhatia**, Head of Department, CSED and **Dr. Seema Bawa** for their support alongwith **Dr. Inderveer Channa**, P.G. Coordinator, for the motivation and inspiration that triggered me for the thesis work.

I would also like to thank the staff members and my colleagues who were always there in the need of the hour and provided with all the help and facilities, which I required, for the completion of my thesis.

Most importantly, I would like to thank my **Family, Friends** and the **Almighty** for showing me the right direction out of the blue, to help me stay calm in the oddest of the times and keep moving even at times when there was no hope.

Monika Dhariwal

80731014

Abstract

Formal Specification of a system has been an active area of research since past few decades. In software engineering, the formal specification of requirement phase is of most importance to achieve rigorous development and maintenance of software systems. Conventional software engineering based on in formal or semi-formal methods are facing challenges in ensuring software productivity and quality. Formal methods have attempted to address those challenges by introducing mathematical notation and calculus to support formal specification, refinement and verification in software development. Despite of their theoretical potential in improving the controllability of software process and reliability, formal methods are difficult to apply to large scale and complex systems in industry due to various practical constraints. For example, limited expertise, time and budget restrictions, etc. Since formal methods are based on mathematics and logic, they can be used for specifying and verifying both hardware and software systems. By using the formal methods, one can minimize the ambiguity and uncertainty inherent in natural language specifications. Once specified formally, a design can be mathematically verified against user requirements. By making this verification process, the implementation of complex and safety-critical system becomes much more credible.

This thesis report suggests which formal specification language is suitable for particular type of problem e.g. communication type problems, real time application, and problems involving concurrency etc. There are a number of formal specification languages, Z, VDM, OCL, SDL etc. This thesis report basically compiles analysis of various formal specification languages and case studies, based on which the best suited formal language for a particular area of application is concluded.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

One of the major problems with the software intensive system is the inadequacy of the system and software specification. These specifications are written in a document known as System Requirement Specifications (SRS) [1]. The requirement document usually defines the major functions of the system, but various details which should be drawn out, cleared up, and solidified in a more detailed specification are not addressed. This causes propagation of inconsistencies and misinterpretations in later stages of design and implementations, and these are often not recognized until the integration stages of system. The cost to fix specifications flaws detected at later stages in software development life-cycle is much greater than if the flaws are detected and fixed at the specification stage itself. Hence there is a strong push to produce more precise and consistent specifications.

There are software requirements development methods based on graphical techniques such as data flow diagrams (DFDs), finite state machines(FTMs), and entity relationship(E-R) diagrams which assist the software engineer in developing better specifications, but these lack precision in the details of specification, the ability to reason about the specification, and developing a smooth path from design to implementation [2]. The formal specification methods overcome these basic objections. They specify the system precisely, the specification can be reasoned about, and they provide a smooth path from design to implementation. Their inherent preciseness(1) remove ambiguity (2) allow one to examine inconsistencies, and (3) guarantees the stability of a system to be implemented. Formal specification languages are generally

concerned with specifying safety properties i.e. properties that state that something 'bad' doesn't happen [3].

Formal methods are further categorized into formal specification and formal verification. Formal specification describes requirements for a system, its operational environment, and design using mathematical logic. Requirement and design specification are two different forms of formal specification. A requirement specification specify what a system is expected to do whereas a design specification rigorously explains how a system is to be built. Formal verification proves completeness or the absence of self-contradictions, and checks whether a design specification satisfies all the requirements by using proof methods available in mathematics and logic.

We can also define formal method as a method that provides a formal language for describing a software artifact (e.g. specification, design, source code) such that formal proof is possible, in principle, about properties of the artifacts so expressed. The above offered definition has two essential components: First, formal methods involve the essential use of a formal language. A formal language is a set of strings over some well-defined alphabets. Rules are given for distinguishing those strings defined over alphabet that belong to the language from strings that do not. Second, formal methods in software support formal reasoning about formulae in the language. These methods of reasoning are exemplified by formal proofs. A proof consists of a sequence of well-defined formulae in the language in which each formula is either an axiom or derived by an inference rule from previous formulae in the sequence. Formal methods are merely an adoption of the axiomatic method, as developed by these trends in mathematics, for software engineering.

1.1 Motivation and Objective

There are a number of formal languages (e.g. Z, VDM, OCL, SDL etc) for software specifications. The software specifications are generally written in natural language. But specifications written in nautral language are

ambiguous, incomplete, and inconsistent. Formal methods seemed to be promising in reducing the chance of error, and hence producing the high quality software system [4]. Despite extensive development over many years and significant demonstrated benefits, formal methods are still poorly accepted by industrial practitioners. Keeping all these things in mind, the review of following domains was done to meet the objective, and phrase the problem for thesis

- Requirements Engineering
- Informal, Semi-formal Specification Language
- Formal Methods

Requirement engineering is concerned with identification, collection and representation of the user's requirements of system. The representation of the software requirement can be in informal language or using formal methods.

Informal specification language includes text and recording in natural language, as well as pictures and animations. "A picture is worth a thousand words." The requirements such expressed can be understood by all stakeholders, including non-technical customers. However, they are very prone to inconsistencies, incompleteness and misunderstanding. These are good for specifying large software systems.

Semi-formal languages consist of Entity relationship diagrams (ERDs), Data flow diagrams (DFDs) and State transitions diagrams (STDs). They are easy to understand, and provide a good overview of the system. Such languages represent a middle way between complete informality and complete formality.

Formal Methods have a great potential as a powerful means for specification of software. They have been successfully applied in several industrial applications, and in certain domains. They are even becoming integral components of various standards. It is useful to capture the specifications in a formal notation rather than the informal specification.

The advantage of formal specification is that it can be checked for consistency, and completeness for some extent, by using the tool available. In this thesis, the aim is to provide a framework for effective use of formal methods in early phase of requirement engineering. Since there are a number of formal specification languages (e.g. Z, OCL, VDM, SDL, LARCH etc), the motivation of the thesis is to provide an approach to identify the most suitable formal specification language for a particular area of application. Another motivating factor for the thesis is the amount of reduction of rework required when the formal methods are used early in development lifecycle, right from the requirement phase.

1.2 Organization of Thesis

Chapter 2 gives an introduction to the various domains described in section 1.1. The information presented in chapter 2 acts as a prerequisite for understanding of rest of the thesis. The chapter is divided into 3 sections, each addressing a particular domain namely. Section 2.1 is concerned with requirement engineering. It shows how critical the phase is in software system development lifecycle and how and to what extent it can affect the success of a software system. Then various requirement engineering processes and methods are described. In section 2.2, the introduction to role of formal methods in software development lifecycle. Formal methods for system specification are introduced in section 2.3. The section describes how formal methods seem to be promising in better software development. The section also gives an overview of some of the existing formal methods.

In chapter 3, the research work in the area of formal language specification and related domain is discussed. It describes various formal languages, their application areas, and tool available for them in detail.

Chapter 4 discusses the problem and objective of the thesis.

Chapter 5 describes the results and analysis of the work done in detail.

Chapter 6 concludes the thesis work. Directions and guidelines for the future work are also discussed in this chapter.

CHAPTER 2

BACKGROUND INFORMATION

This chapter gives an introduction to various domains that have been investigated for the completion of the thesis.

2.1 Requirement Engineering

Requirements describe the desired functionality of a system. In general, there are two types of requirements: functional and non-functional. Functional requirements describe the behavioral aspects of a system whereas non-functional requirements describe the non-behavioral aspects. Requirement Engineering is the initial and a critical phase of system development lifecycle [5, 6]. The requirement engineering process influences the overall success of the project. The requirement phase is an important phase of the project. Even a few inaccuracies or misunderstanding in requirements can increase the overall development and test effort. Requirement error found late in development lifecycle can create weeks of rework by developers and testers [7].

As customer satisfaction is becoming the predominant measurement of a system's quality, correctly understanding, documenting and validating the needs of the stakeholder involved in the development become more and more crucial. The correctness of product requirements is the key point in determining the success or failure of project development. The product requirements are written first as a specification.

A program specification is a statement which describes

- what purpose the program serves and
- how the program can be correctly used [8]

A software system often has defects, and defects do cost. The cost can be in terms of money, human life or integrity of critical infrastructures.

The requirement engineering phase is the most common point of introduction of defects. Considering that requirements errors are the most numerous, as well as the most costly and time-consuming errors, it is very surprising that not more effort is put into the requirements specification. Studies show that only 6% of the project costs and between 9% and 12% of the project duration are spent in the requirements phase.

Surveys have concluded that 50 – 80% errors are introduced right from the requirements phase. The cost of detecting and correcting errors at later stages of development cycle increases exponentially [6]. This may be inferred from above data that the cost of defects may be decreased substantially by improving the requirement engineering activities.

Requirement Engineering consists of the following activities/tasks [9, 10]:

• **Requirement Elicitation**: Requirements elicitation has been defined as "The process of identifying needs and bridging the disparities among the involved communities for the purpose of defining and distilling requirements to meet the constraints of these communities" [6]. In requirements elicitation, there is almost always a need for the participation of stakeholders (e.g. domain experts, customers and end-users) to help requirements engineers to uncover requirements. It also requires a good deal of customer contact and domain knowledge. Identification of the stakeholders is also an essential task in requirements elicitation. The requirements elicitation also includes context analysis, which addresses the purpose of the system – why the system is being developed at all. The techniques used for requirements elicitation include interviews, use cases and scenarios, discussions, simulation and prototyping.

• **Requirement Analysis**: Once the requirements have been gathered, the next phase is requirement analysis. It is the categorizations of requirements

and their organization in related subsets, the relationship among the requirements, examining the qualities of requirements like consistency, unambiguity and completeness. All this is done based on the need of the customer. One more important task during the requirement analysis phase is feasibility analysis, i.e. are the requirements implementable or not. This can be done by negotiating with the customer and ranking the requirements. We try to retain at this stage what the customer needs rather than what he wants. Past experience can be used for the purpose. It helps to reduce the amount of complexity that must be comprehended at one time, is inexpensive to build compared to the real thing, and facilitates the description of complex aspects.

• **Requirement Specification:** Requirement specification aims at the production of an SRS (Software Requirement Specification). The main purpose of a requirements document is to convey information gathered from the customer and other sources to the developer [11]. A specification can be a written document, a graphical model, a formal or mathematical model, mental model or a combination of these. Requirement specification languages can be categorized into following classes:

i) Informal languages: Text and recordings in natural language, as well as pictures and animations, fall into this category. They are very expressive. The requirements such expressed can be understood by all stakeholders, including non-technical customers. However, they are very prone to inconsistencies, contradictions, incompleteness and misunderstandings [6]. These are good for specifying large software systems.

ii) Semi-formal languages: Entity relationship diagrams (ERDs), data flow diagrams (DFDs) and state transition diagrams (STDs) are very commonly used within industry for the semi-formal expression of requirements. They are easy to understand and provide a good overview of the system. Such languages represent a middle way between complete informality and complete formality, and can be used

for the transition from informal requirements to a formal specification [6].

iii) Formal languages: Formal languages have been introduced for quite a while but they are not very common in practice. They have the advantages such as conciseness, completeness, automated reasoning, etc. The examples of formal languages include Z, OCL and VDM. These are based on the use of mathematical notations.

The selection of a category of specification languages and a language there of is a cumbersome task. It is generally preferable and advisable to use a blend of these languages. The degree of precision of the requirement document should be based on the criticality of getting a chunk of software right the first time. Where mathematics is included in the requirements, the document must also include English text to explain the mathematics [7] . This thesis addresses the issue of integrating informal and formal languages for the purpose.

The use of a standard template is sometimes suggested, so that the requirements are consistently represented. However, it can place restrictions on specifying the system, and sometimes, some concepts may not be altogether possible to be described using that template.

- **Requirements Validation:** The purpose of requirements validation is to certify that the specified requirements comply with the given user and customer intentions. This means that the requirements need to be expressed in a notation that is understandable by the customer. Suitable techniques for validation are prototyping, scenarios, checking of specifications against domain models, natural language paraphrasing, animation, simulation, etc. The requirements validation can be done by using reviews. The requirements are to be reviewed by various stakeholders i.e. users, developers, designers, domain experts and requirements engineer. In requirements validation, requirements can be examined against a checklist questions [10].

• **Requirements Management:** Requirement management is the set of activities carried out to identify, control and track requirements and changes to requirements. During requirements engineering, system development and operation, new requirements are discovered and current requirements are changed. This evolution of requirements throughout the whole software development life cycle has to be managed in order to ensure high-quality specifications. Although requirements management may look like an overhead in the beginning, it is usually rewarded by better customer satisfaction and lower overall system development costs [6]. The main reason for the need of requirements management is the volatile nature of requirements. The volatility of requirements means that requirements change frequently. The changes can be due to errors or misunderstandings in requirement specifications or problems in design and implementation of the requirements. The reasons for change in requirements can be:

i)Change of Scope: A change of scope of system may be caused if the user has changed his mind, or a newer process has been adopted. This may also be caused due to non-feasibility of implementation of certain requirements. The interfacing of requirements with other requirements is also a cause for change of requirements.

ii)Change in External Interfaces: The requirements may be changed, addedor removed as a result of change in external interfaces of the systems, such as political, social, laws of government or the environment of the system.

iii)Poor Understanding: The requirements may change if the customers/users are not completely sure of what they need. Another cause can be that a technical person dealing is not well versed with domain knowledge. The omission of some 'obvious' requirements at the earlier stages can also invite a change in requirements.

• **Requirements Traceability:** Requirement traceability is generally considered as a part of the requirements management process. It is about

finding the relationships of requirements with the system aspects. The traceability tables can be developed for the purpose. Some types of the traceability are described below.

i) Requirements-sources traceability: Links the requirement with the people or documents that specified it.

ii) Requirements-rationale traceability: Links the requirement with a description of the rationale for it. It is concerned with the sensibility of the requirement specified.

iii)Requirements-requirements traceability: Links the requirement with other requirements that depend on it, and allows the creation of a requirements hierarchy. It is also known as requirements dependency traceability.

iv) Requirements-architecture traceability: Links the requirements with the interfaces of external systems that are used in the provision of the requirement. This is important where there is a high dependency on other systems.

The requirement engineering is an important and necessary phase of the software system development. If this is done carefully, it can result in reduced development time and cost, higher software quality and higher customer satisfaction. A poor requirement engineering process, on the other hand, can lead to systems that are inadequate, incomplete, erroneous, behind schedule and over-budget, and do not meet the customers' expectations.

2.2 Role of Formal Methods in software development lifecycle

Using formal methods in software development is an important step towards achieving correctness, consistency, and understanding in software development process. Formal methods are used in all phase of system development [4]. Such application should not consider these a separate

activity, but rather an integral one. Consider, for each system development phase [12]:

I. Requirement Analysis- In the requirement analysis phase, requirements are often specified informally with a language in which the end-user or customer is familiar. This often results in natural language (e.g. spoken and written English) to create a requirement document such as system requirement specification (SRS) or concept of operations. Such informal specifications are not adequate in that they are often inaccurate, inconsistent, and ambiguous. Natural language is also very lengthy, making them difficult to check for completeness. Applying a formal method helps clarify a customer's set of informally stated requirements. A specification helps crystallize the customer's vague ideas and reveals contradictions, ambiguities and incompleteness in the requirements.

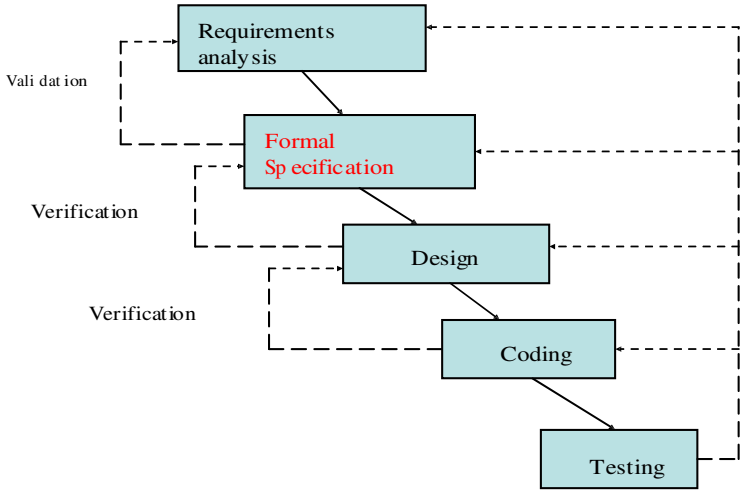

Fig. 2.1:Use of formal languages in SDLC [12].

II. System design-Two of the most important activities during design are decomposition and refinement. The Vienna Development Method (VDM),

Z, and Lamppost's transition axiom method are formal methods that are especially suitable for system design. An alternative approach to program development based on constructive logic gave rise to proof development environments like Nuprel in which programs are proofs and vice versa.

III. System verification—Verification is the process of showing that a system satisfies its specification. Formal verification is impossible without a formal specification. Systems such as Gypsy, the Hierarchical development Method (HDM), the Formal Development Methods (FDM), and m-EVES (Environment for verifying and Evaluating Software) evolved as a result of a primary focus on program verification. Higher Order Logic (HOL) was originally developed for hardware verification.

IV. System validation—Formal methods can aid in system testing and debugging. Specification alone can be used to generate test cases for black-box testing. Specification that explicitly state assumptions on a module's use identify test case for boundary conditions. Only a few formal methods have been developed explicitly for testing. The examples are the data Abstraction, Implementation, Specification, and Testing System, used to test implementation of abstract data types; Kemmerer's symbolic execution tool, used to generate and execute test cases from Ada to specifications.

V. System Documentation—A specification is a description alternative to system implementation. It serves as a communication medium between a client and a specifier, and among team members of the implementation team. In reply to question "What does it do?" no answer is more exasperating than "Run it and see." One of primary intended uses of Formal method is to capture the "WHAT" in formal specification rather than the "HOW."

2.3 Formal Methods

The Encyclopedia of Software Engineering defines formal method in the following manner: "A method is formal if it has a sound mathematical basis, typically given by a formal specification language. This basis provides a means of precisely defining notions like consistency and completeness, and more relevantly, specification, implementation and correctness"[10].

Formal specification of systems has been an active area of research for quite some time [13]. Since the requirements document forms the basis of the whole development process, such defects can have severe consequences for the whole project. Therefore, it is important to deal with these defects in a requirements specification right from the start. The requirements document must be at the very least correct, complete, and unambiguous [14]. Formal specification methods, strive to achieve these properties of the requirement specification document.

Formal methods, put in simple words, means to use mathematical techniques in software specification, design and construction. We can say here that mathematics is important in software engineering, as it is in other disciplines of engineering. Formal methods are, in contrast with natural language, unambiguous and precise in nature. Formal methods have been advocated as one way to define requirements with mathematical precision and rigour. Traditionally, specifications have been written in natural language, but today more and more specifications are written in formal specification languages [15]. They are used regularly while verifying systems that are of critical importance or real time event driven, where errors are not acceptable [16].

However, formal methods just specify what the system is to do and not how it must be done. A formal specification is a statement in a formal specification language. A formal specification is usually written in a concrete language. If this language has a precisely defined syntax and

semantics, a specification written in it is formal. They provide the means to build system models by specifying their structure and behavior. When used in conjunction with tools support, automated verification of properties for complex models is also possible.

Formal specifications have the additional advantage over informal ones of being amenable to machine analysis and manipulation [8]. Formal methods allow one to be precise about the functional properties of a software system before the implementation details' of that system have been worked out. A formal specification:

- contains information as detailed as the code itself
- is from a users perspective
- is easier to understand and reason about than a system design or the source code.

Formal methods are convenient for specifying large software systems, where completeness and consistency are important issues [17]. The formal methods can be used

- To clarify understanding of the problem
- To communicate intentions to other developers
- To provide a prototype to demonstrate ideas
- To use as a basis for design (formal or informal)
- To prove the final software correct
- To explore mathematically the consequences of the specification [18]

Following recommendations regarding the actual introduction of formal methods into an established project are made:

i) Seek commitment from all levels: management to system engineers.

ii) The specification must be written by system engineers. A specialist in the formal notation is needed on the project but this person should serve an advisory role only. The system engineers with application knowledge must be the ones writing the specification.

iii)Specify a portion of the system that does not require a complicated state model first. This allows the group to focus on specifying operations early.

iv)Start with a small project. Early success on the small project will help gain buy-in for the formal method process.

v)Spend time deciding what should, and what should not, be formally specified. Specify only those portions of the system that need clarification.

vi) The language should be used with flexibility at first. Understanding is the primary concern. Making the specification precise enough to support proofs should be secondary at this stage.

In addition to these recommendations, there are 'The Ten Commandments of Formal Methods' as described [10, 19] :

1)Thou shalt choose an appropriate notation: To choose effectively from the wide array of formal specification languages, some of which are discussed later in this section, a software engineer should consider the application to be specified and the breadth of usage of language.

2)Thou shalt formalize but not over-formalize: It is generally not necessary to apply formal methods to every aspect of a major system. Those components that are safety critical are first choices, followed by components whose failure cannot be tolerated..

3) Thou shalt estimate costs: Formal methods have high startup costs like training of staff, cost of tools etc.

4) Thou shalt have a formal methods guru on call: Expert training and on-going consulting is required when using formal methods for the first time.

5) Thou shalt not abandon thy traditional development methods: Formal methods should be used as an aid to the existing conventional methods.

6)Thou shalt document sufficiently: It is recommended that natural language be used along with the formal specification of the system, for reader's understanding.

7) Thou shalt not compromise thy quality standards: Formal methods are not a panacea. There is nothing magical about them. SQA activities must continue to be applied.

8) Thou shalt not be dogmatic: Absolute correctness in the real world can never be achieved. Mathematical models can be verified with a good level of certainty, but these models might not correspond with reality correctly. When applying formal methods, the level of use should always be determined beforehand and monitored while in progress. A project manager should always be prepared to adjust the level of use if required.

9) Thou shalt test, test, and test again: Formal methods will never replace testing; rather they will reduce the number of errors found through testing. Formal development and testing tend to avoid and discover different types of error, so the two are complementary to some extent.

10)Thou shalt reuse: Software reuse helps in reducing cost and increasing of a software system. Formal methods can be a good approach for construction of software libraries.

Having discussed the guidelines for the use of formal methods, I would also like to state some of the myths related to formal methods [19, 20]:

Myth 1: Formal Methods can guarantee that software is perfect: Any technique is fallible, and formal methods are no exception. Even if a correct mathematical proof is achieved, the assumption that the mathematics can model reality correctly is still prone to error.

Myth 2: They work by proving that programs are correct: It is not necessary to undertake proofs to gain benefit from the use of formal methods; indeed much if not most industrial use of formal methods does not involve proofs. Major gains can be achieved just

by formally specifying the system being designed since this process alone can expose flaws, and in a much more cost effective manner. Proofs may be worthwhile in highly critical systems where the extra cost can be justified.

Myth 3: Only highly critical systems benefit from their use: A range of formal methods have been applied to many types of system, some of greater, others of lesser criticality. The extent and type of application will depend on the level of criticality, which is ultimately a case of engineering and financial judgement.

Myth 4: They involve complex mathematics: The mathematics required for formal specification is of a level that could be taught at school. After all, a major goal of a specification is to be easily understandable, so using esoteric terminology is in nobody's interest. Unfortunately, although relatively simple, it is a fact that many software engineers have not received the requisite training in the past.

Myth 5: They increase the cost of development: Proofs do increase the cost of development in general, but formal specifications do not if used appropriately. This is because they allow many errors to be discovered earlier on in the design process when they are still relatively cheap to correct.

Myth 6: They are incomprehensible to clients: The mathematics may not be readable by an untrained client, but a formal specification helps produce a much clearer natural language description of the system as well. This should be presented to the client, giving a much less ambiguous description of the system than is often the case.

Myth 7: Nobody uses them for real projects: There are now a number of examples of actual use of formal methods, with demonstrably beneficial results. Two recommended examples which used Z, and both of which won UK Queen's Awards for Technological Achievement in 1990 and 1992, are the Inmos Transputer Floating Point

Unit microcode design; and the IBM CICS Transaction Processing System.

Myth 8: Formal methods delay the development process: Some projects using formal methods have been seriously delayed in the past, but this has been as much to do with the problem of introducing any new technique into the design process as to do with formal methods.

Myth 9: They do not have tools: There are now some significant tools supporting formal methods, many of which have been put to serious industrial use. Examples include RAISE (Rigorous Approach to Industrial Software Engineering), Larch Prover etc.

Myth 10: Formal methods replace traditional engineering design methods: Formal methods should not be used to replace the existing development process. Rather they should be slotted into the process in an appropriate and thoughtful manner. Formal methods can also be used effectively to augment an existing design process by providing extra feedback to correct errors early in the design process.

Myth 11: They only apply to software: Formal methods are used for hardware development as well as software. The Inmos Transputer work mentioned in Myth 7 is one example. Z has also been applied to microprocessor instruction sets and oscilloscopes etc.

Myth 12: Formal methods are unnecessary: Although there are occasions in which formal methods are in a sense overkill, in other situations, they are very desirable. In fact, the use of formal methods is recommended in any system where correctness is of concern. This clearly applies to safety and security critical systems, but it also applies to system in which ensuring the avoidance of catastrophic failure is desired.

Myth 13: Formal methods are not supported: There are now many books on formal methods. A number of companies now specialize in formal methods. A range of formal methods tools are commercially marketed.

Myth 14: Formal methods people always use formal methods: While formal methods can be useful, they are not always appropriate. Even those well versed in the use of formal methods do not always use them.

Basically the use of term Formal method is to describe any approach which utilises a formal specification language and specifies the role of that formal specification during the software development process. In other words, the use of formal method will not necessarily imply any formal refinement process, formal reasoning or proof. The term formal specification will then be used to refer to any language in which it is possible to specify fully the functionality of the whole or part of a piece of software in a way amenable to formal reasoning. The emphasis will be on the fact that it is based on a firm mathematical basis, rather than whether it is sufficiently abstract. So, formal methods can provide:

- More precise specifications
- Better internal communication
- An ability to verify designs before executing them during test
- Higher quality and productivity

2.3.1 Drawbacks of semi- formal and informal language

Applying semiformal methods that emphasize the use of graphical representations of the software. A major problem with the semiformal approach, however, is the lack of precise semantics, which may lead to ambiguous interpretation of certain requirements [12].

The major problems with Informal specifications are following:

- Informal specifications are likely to be ambiguous, which is likely to cause misinterpretation.
- Informal specification is difficult to be used for inspection and testing of programs because of the gap between the functional descriptions in the specification and the program structures.
- Informal specifications are difficult to be analyzed for their consistency and validity.

- Informal specifications are difficult to be supported by software tool in their analysis, transformation, and management (i.e. search, change, and reuse).

2.3.2 Advantages of formal methods

The advantages of using math for any analytical problem are [12]:

- Short notation.
- Forces you to be precise
- Identifies ambiguity.
- Clean form of communication.
- Makes you ask the right questions.

2.3.2.1 Short notation

By using the Formal specification, we can shorten the written text. For example, (without Formal specification language) "For every ticket that is issued, there has to be a single person that is allowed to enter. This problem is called the owner of the ticket."

With Formal specification language (By using Z notation):

Ticket Owner: Issued Tickets-> Person

2.3.2.2 Force Precision

The Formal specification tells us about the following ones:

- How should we build a system based on the given requirements?
- What will happen in the exceptional case (Formalization fail)?

2.3.2.3 Identifying Ambiguity

Formal methods are also useful in identifying the ambiguity as follow: For example, (without use of formal specification language) "When temperature is too high, the ventilation has to be switched on or the maintenance staff has to be informed." But by using the formal specification, we may do both as follow:

Temperature Is High=> (Notifying staff or ventilation on)

2.3.2.4 Clear form of communication

Every mathematical notation has a precise semantic definition.

New constructs can be added defined in terms of old constructs.

Mathematics does not need language skills and can be easily understood in an international context.

2.3.3 Way of writing Formal specification language

A variety of formal specification languages are available. Following paragraphs give a brief overview of the formal specification languages commonly in use, which are deeply described in next chapter.

• **Z:** Z (pronounced as zed) makes use of the set theory and discrete mathematics. It has been developed at the Oxford University Computing Laboratory (OUCL) by the Programming Research Group. The specifications written in Z can be read more easily by human beings, than by computers. It is more readable than other formal languages. However, human beings too need to be specially trained to read and understand Z notation and thus the formal specifications written using Z. Z is a typed language; that is to say, every variable in Z has a particular type (i.e., set from which it is drawn) associated with it which must match appropriately when it is combined with other variables.

• **OCL (Object Constraint Language):** OCL is a formal notation developed so that the users of UML can add more precision to their specifications. All of the power of logic and discrete mathematics is available in the language. However, the designers of OCL decided to use only ASCII characters, rather than conventional mathematical notation. This makes the language to be more easily processed by

computer and editing the specifications easier. But it also makes OCL a little wordy in places.

• **VDM (Vienna Development Method):** VDM is a model-oriented formal specification and design method based on discrete mathematics. The underlying formal specification language is known as META-IV. It has been used to develop compilers, databases, fault-tolerant systems, security-critical message-processing systems, etc.

• **SDL (Specification and Description Language):** SDL is a specification and description language with a textual as well as a graphical representation. It is based on communicating extended finite state machines, and has object-oriented features. Although it has mostly been used for telecommunications services and protocols, it is now also being applied to aircraft and train control, as well as packaging systems.

• **LARCH:** LARCH is a specification language supporting educational theories embedded in first-order logic. The Larch Prover (LP) is an interactive theorem-proving system working midway between proof-checking mode and fully automatic theorem-proving mode. It has been used in hardware design and verification, concurrent algorithms, etc.

The detailed study of these formal languages will be done in next chapter, that is, what type of syntax they use, what are the data type and how they are used.

CHAPTER 3

LITERATURE REVIEW

In this chapter we review different type of formal languages proposed by researchers. The choice of a method is likely to affect what a specification says and how it is said. A method's guidelines may encourage the specifier to be explicit about some system behaviors (for example, state changes) and not others (for example, error handling). Syntactic convention (such as indentation style), special notation (vertical and horizontal lines), and keywords affect a specification's physical appearance and its readability. But, in a more practical sense, formal methods do not differ radically from one another. Within some well-defined mathematical framework, they let system developers couch their ideas precisely. The more rigor applied in system development, the more likely developers are to state requirements correctly and to get the design right and, of course the more precisely they can argue the correctness of the implementation.

3.1 Z Notation

Z (pronounced as zed) makes use of the set theory and discrete mathematics. It has been developed at the Oxford University Computing Laboratory (OUCL) by the Programming Research Group. Z can be used to model static as well as the dynamic aspects of the system. The static aspects include data invariant, i.e. a condition that is same throughout the execution of the program; and dynamic aspects include the operations and the change that happens in the state of the system. Fig. 3.1 shows the Z schema format.

Where:

The **Schema** – a means for structuring a formal specification.

The **declaration** gives the type of Function or constant.

The **Predicate** gives its value.

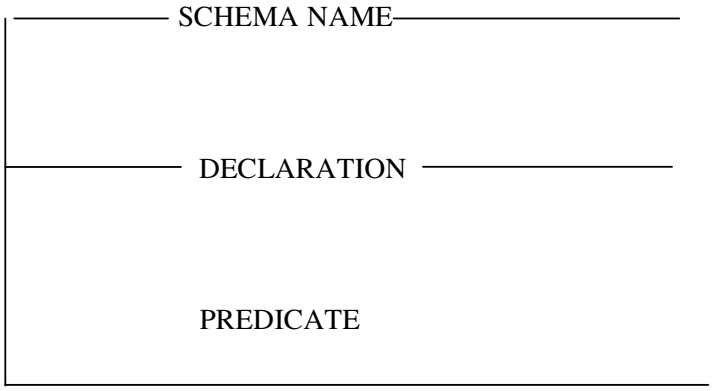

Fig. 3.1: Generic structure of schema [10]

3.1.1 Formal methods concepts in case of Z notation

Data Invariant-A data invariant is a condition that is true throughout the execution of the system that contains a collection of data

State-In Z specification, the state of a system is represented by the system's stored data (hence, Z suggests a much larger number of states, representing each possible configuration of data).

Operation-Operation is an action that takes place within a system and read or writes data. Three types of conditions can be associated with operations:

I. **Invariant**- an invariant defines what is guaranteed not to change.
II. **Precondition**- a precondition defines the circumstances in which a
III. **Postcondition**- postcondition of an operation defines what is guaranteed to be true upon completion of an operation. This is defined by its effect on data [21].

3.1.2 The mathematics symbols used in Z notation

As Z uses the set theory and logics, here is a brief description of these symbols [10].

Sets

S: P X	S is declared as a set of Xs.
S v T	The union of S and T: it contain every member of S, T.
S ^ T	Intersection of S and T, contain common member of S and T
$	Empty set: it contains no members.
{X}	Singleton set: it contains only one member.
N	The set of natural numbers 0, 1, 2, 3…….....
S: F(X)	S is declared as a finite set of Xs.
Max (S)	The maximum of the nonempty set of numbers S.

Function

f: X -> Y	f is declared as a partial injection from X to Y.
Dom f	**The domain of f** : set of value of X for which f(x) is defined.
Ran f	**The range of f:** set of values taken by f(x) as X varies over domain of f

Logic

P^Q	**P and Q:** it is true only if both P and Q are true.
P=>Q	**P implies Q :** it is true if either Q is true or P is false.
* S =* S'	No components of schema S change in an operation.

3.1.3 Example of Z

Birthday book is a system which records people's birthdays, and is able to issue a reminder when the day comes round. For this we shall need to deal with people's names and with dates. For present purposes, it will not

matter what form these names and dates take, so we introduce the set of all names and the set of all dates as basic types of the specification [21]:
[NAME, DATE].

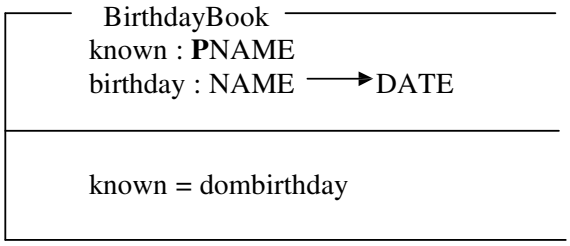

Fig. 3.2: Schema of Birthday Book

Where
known is the set of names with birthdays recorded;

birthday is a function which, when applied to certain names, gives the birthdays associated with them. In this example, the invariant allows the value of the variable known to be derived from the value of birthday: known is a derived component of the state, and it would be possible to specify the system without mentioning known at all. One possible state of the system has three people in the set known, with their birthdays recorded by the function birthday:

known = { John, Mike, Susan }
birthday = { John → 25–Mar,
Mike → 20–Dec,
Susan → 20–Dec}.

The invariant is satisfied, because birthday records a date for exactly the three names in known. There are a number of operations, for example AddBirthday, FindBirthday, Remind etc. Here we explain the operation AddBirthday: The declaration ΔBirthdayBook alerts us to the fact that the schema is describing a state change: it introduces four variables known,

birthday, known' and birthday'. The first two are observations of the state before the change, and the last two are observations of the state after the change.

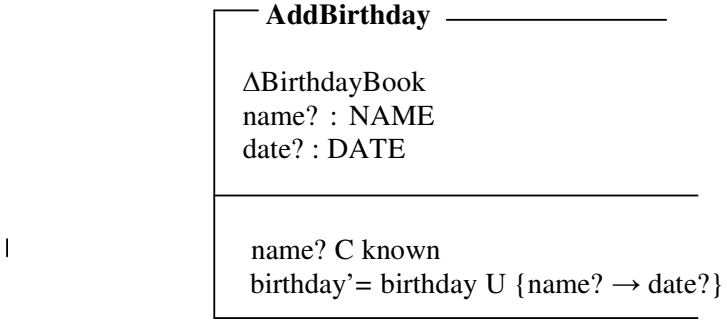

Fig.3.3: Structure of AddBirthday

Each pair of variables is implicitly constrained to satisfy the invariant, so it must hold both before and after the operation. Next come the declarations of the two inputs to the operation. By convention, the names of inputs end in a question mark. The part of the schema below the line first of all gives a pre-condition for the success of the operation: the name to be added must not already be one of those known to the system. This is reasonable, since each person can only have one birthday.

3.1.4 Tools Available

ZTC is a type-checker for Z specification notation. It determines if there are syntactical and typing errors in Z specifications**.** ZTC accepts two forms of input: LaTeX with oz or zed packages, and ZSL.

The fuzz package is a syntax and type-checker with a LaTeX style option and fonts. It is available from the Spivey Partnership and is compatible with the 2nd edition of the Z Reference Manual.

Zola is a commercial integrated support tool for Z on Sun workstations, for automated assistance at all stages of the specification construction, proving and maintenance process. It is intended for system developers and includes a WYSIWYG editor, type-checker and tactical theorem prover suitable for the creation and maintenance of large specifications.

DST-fuzz is a set of tools based on the fuzz package by Mike Spivey, supplying a Motif based user interface for LaTeX based pretty printing, syntax and type-checking [22].

3.2 OBJECT CONSTRAINT LANGUAGE (OCL)

Object Constraint Language (OCL), is a formal language to express side effect-free constraints. The Object Constraint Language (OCL) is an expression language that describes constraints on object-oriented languages and other modeling artifacts. A constraint can be seen as a restriction on a model or a system. OCL is part of Unified Modeling Language (UML) and it plays an important role in the analysis phase of the software lifecycle. OCL is the expression language for the Unified Modeling Language (UML). To understand OCL, the component parts of this statement should be examined [23]. Thus, OCL has the characteristics of an expression language, a modeling language and a formal language.

1) Expression language- OCL is a pure expression language. Therefore, an OCL expression is guaranteed to be without side effect. It cannot change anything in the model. This means that the state of the system will never change because of an OCL expression, even though an OCL expression can be used to specify such a state change.

2) Modeling language- OCL is a modeling language, not a programming language. It is not possible to write program logic or flow-control in OCL.

3) Formal language- OCL is a formal language where all constructs have a formally defined meaning. The specification of OCL is part of the UML specification. OCL is not intended to replace existing formal languages, like VDM, Z etc.

3.2.1 OCL notation and operations

OCL provides built-in operations; a small sample is presented in table 3.1.

Table 3.1: Summary of key OCL notation [10].

OCL notations	Meaning
X.Y	Obtain the property Y of object X. A Property can be an attribute, the set of objects at end of an association, or other things depending on the type of UML diagram.
C f()	Apply the built-in OCL operation f to collection C itself (as opposed to each of the objects in C).
P implies q	True if either q is true or p is false.
C size()	The number of elements in collection C.
C isEmpty()	True if C has no elements, false otherwise.
C1 includesAll(C2)	True if every element of C2 is found in C1
S1 intersection(S2)	The set of those elements found in S1 and also in S2.
S1 union(S2)	The set of elements found in either S1or S2.
S1 excluding(X)	The set S1 with object x omitted.

3.2.2 Exapmle of OCL

Block Handler: Block Handler, for example the very first step for using OCL is to develop a UML model (class diagram), which specifies many relationship among the objects involved [10].

1. No block will be marked as both used and unused.

Context BlockHandler inv: self.used intersection(self.free)) isEmpty()

Here, **context** indicates the element of the UML diagram that the expression constrains. The keyword **self** refer to the instance of object (in this case instance of BlockHandler). If **self** is class **C**, with attribute **a**, then **self.a** evaluates to the object stored in **a**. **Self.a** defines the 'a' attribute or property of object.

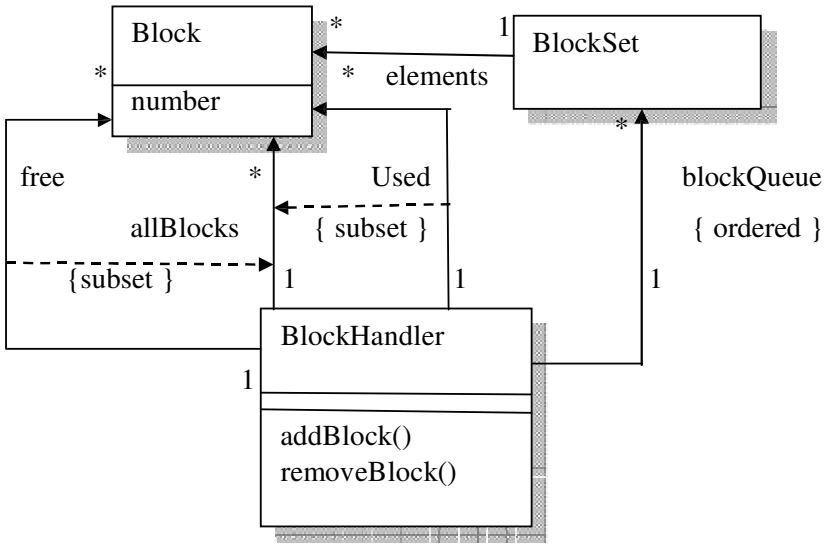

Fig.3.4: Class Diagram for a Block Handler [10]

2. All the sets of blocks held in the queue will be subsets of the collection of currently used blocks.

Context BlockHandler inv:

blockQueue forAll(aBlockSet I used includesAll(aBlockSet))

One of the plus point of using OCL is that it is more friendly to people who are less mathematically inclined. Also it is more easily processed by computer.

3.2.3 Tools Available

Currently, a number of tools that support OCL are available, from both universities and commercial companies, which are briefly described here:

Octopus, the "OCL Tool for Precise UML Specifications". Octopus is an Eclipse plug in. by IBM: which is able to check the syntax of OCL expressions, as well as the types and correct use of model elements like association roles and attributes [24].

XMF-Mosaic from Xactium is a commercial model-based tool that supports MDA and the rapid development of customized MDA tools. XMF-Mosaic includes support for OCL, and an extended form of OCL, called XOCL, which is fully executable.

.

3.2.4 Applications of OCL:

OCL can be used for a number of different purposes:

- To specify invariants on classes and types in the class model
- To specify type invariant for Stereotypes
- To describe pre- and post conditions on Operations and Methods
- To describe Guards
- As a navigation language
- To specify constraints on operations

Traditional formal languages are useable to persons with a string mathematical background, but difficult to use for the average business or system modeler. OCL has been developed to fill this gap. OCL is a formal language, which remains easy to read and write. In object-oriented modeling a graphical model like a class model, is not enough for a precise

and unambiguous specification. There is a need to describe additional constraints about the objects in the model. Such constraints are often described in natural language. Practice has shown that this will always result in ambiguities. In order to write unambiguous constraints, so-called formal languages have been developed. The disadvantage of traditional formal languages is that they are usable to persons with a string mathematical background, but difficult for the average business or system modeler to use. OCL has been developed to fill this gap.

3.3 SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)

Specification and Description Language (SDL) is an object-oriented, formal language defined by The International Telecommunication Union-Telecommunications Standardization sector. The language is intended for the specification of complex, event-driven, real-time, and interactive applications involving many concurrent activities that communicate using discrete signals [25].

The basic theoretical model of SDL system consists of a set of extended Finite State Machines (FSMs) that run in parallel. An SDL system consists of following components:

- Structure—system, block, process, and procedure hierarchy
- Communication—signal with optional signal parameters
- Behavior—processes
- Data—abstract data type (ADT)
- Inheritance—describing relation and specialization.

3.3.1 SDL Characteristics

SDL is a design and implementation language dedicated to advanced technical systems (i.e. real-time system, distributed systems, and generic event-driven systems where parallel activities and communication are

involved). Typical application areas are high-and low- level telecom systems, aerospace systems, and distributed or highly complex mission- critical systems. SDL has a set of specialized characteristics that distinguishes it from other technologies:

- Standard
- Formal
- Graphical and symbol-based
- Object-oriented (OO)
- Highly Testable
- Portable, Scalable, and Open
- Highly Reusable

3.3.2 Example of SDL

LSB (Local switch board) The LSB purpose is to connect users locally, with each other [31]. A user interacts with the LSB through his/her phone. The LSB is a simple one, where phone lines and phone numbers are allocated statically at the system startup in a permanent way. Static requirements of LSB include 9 local lines, numbered from 1 to 9. The signals passed from user phones to LSB are: OffHook, OnHook, and DialedDigit, while the signals passed from LSB to user's phones: DialTone, BusyTone, RingTone, and ConnectTone. When an action is expected from a user, he/she must react within maximum 15 seconds. The LSB system has two blocks one to model the switch and one to model the line handler.

An SDL block is a structural entity that may contain processes (active entities) or other blocks. The two communicate by mutually exchanging messages and the line handler also exchanges messages with the environment. The line handler block contains a process, line that describes the reactive behavior of a line, through a state machine and three procedures: called, connected, and calling. The switch block describes the actual behavior of the switch. Concretely, it is composed of connection agent that monitors a local call and connection

manager that receives connection requests from users and allocates connection agents accordingly. Figure 3.5, depicts the line handler's state machine using the SDL visual notation

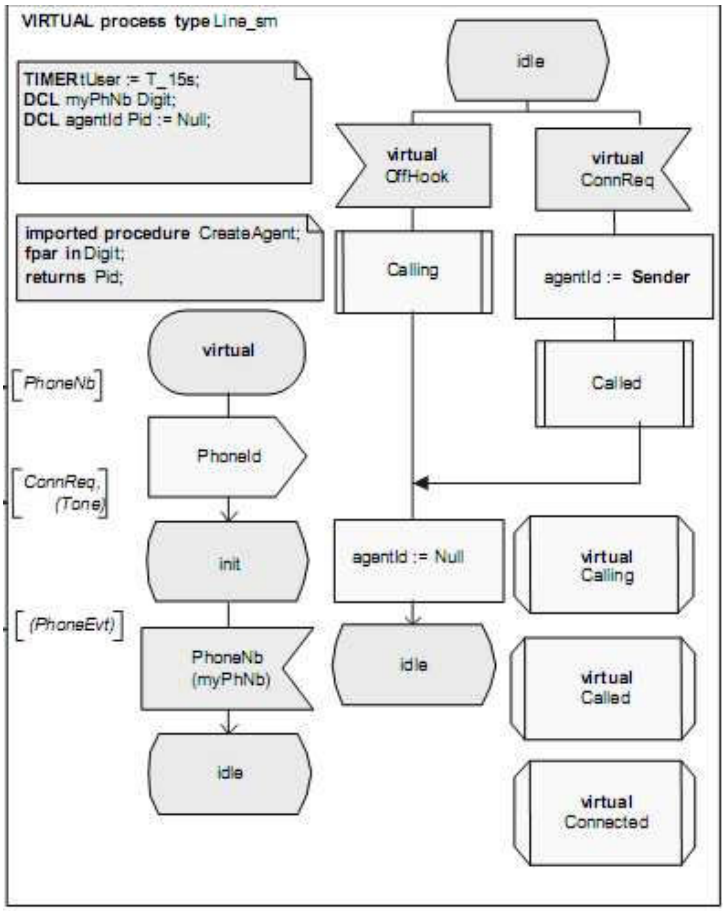

Fig. 3.5: State machine diagram of process line [31]

3.3.3 Benefits of SDL

SDL has been defined to meet the following demands [25]:

- A well-defined set of concepts.
- Unambiguous, clear, precise, and concise specifications.
- A thorough and accurate basis for analyzing specification.
- A basis for determining whether or not an implementation conforms to the specifications.

3.3.4 Application area

Although SDL evolved within telecommunications, it is becoming increasingly popular in other industries as well. Some examples of applications of SDL outside the telecommunication area include the following:

- Satellite communications.
- Aeronautical standardization.
- Medical equipment.
- Railway control system.

3.3.5. Tools Available

Cinderella SDL is a visual modeling tool for developing embedded software systems, communications services, protocols, or any kind of message/signal based system [26].

SanDri-La Visio Add-on provides a complete Visio add-on application for the creation of SDL, MSC and TTCN diagrams. These diagrams are defined by ITU standards Z.100 (SDL-2000), Z.109 (UML support), Z.120 (MSC-2000) and Z.140 (TTCN v3).

SAFIRE-SDL, which is a graphical tool chain designed for the implementation and validation of state machines (FSMs). SAFIRE-SDL uses a graphical representation, based on international standards SDL, MSC, TTCN and ASN.1, in a single environment covering the design, execution and observation of both FSMs and test scenarios. SOLINET also offers libraries for standard signaling systems (ISDN, SS7, V5, GSM, INAP, IP, etc) and a corresponding protocol analyzer.

3.4 VIENA DEVELOPMENT METHOD (VDM)

VDM stands for "The Vienna Development Method": a collection of techniques for the formal specification and development of computing systems. It consists of a specification language called VDM-SL; rules for data and operation refinement which allow one to establish links between abstract requirements specifications and detailed design specifications down to the level of code; and a proof theory in which rigorous arguments can be conducted about the properties of specified systems and the correctness of design decisions [27].

VDM-SL is a model-oriented specification language. This means that a specification in VDM-SL consists of a mathematical model built from simple data types like sets, lists and mappings, along with operations which change the state of the model. For example, a specification of a hotel reservation system would contain a mapping from room numbers to names and addresses of occupants (modeled as character strings), along with operations to add and remove guests and rooms, occupation dates etc. Basic types like natural numbers, characters, and type constructors like sets and maps, are provided "for free" in VDM-SL is used.

The use of VDM in the development of compilers, databases, fault-tolerant storage systems, graphics software, medical warning systems, novel computing architectures, security-critical message processing systems and more. VMD is a formal, mathematically oriented method for specification of systems and development of software. In general, formal method falls into two classes: Algebraic language and Model-Oriented Language. VDM is a model-based method. Its main idea is that of giving description of software system and other system as model. Model –based methods are differ from algebraic and other formal methods in that they explicitly define type of object of concern and utilize primitive, predefined operation in defining high- level operation [28].

At the uppermost levels of system description, Meta-IV, which is the specification of language of VDM, is used for expressing the model. The models are defined using a number of type definitions (for the objects) and function definitions (for the operations). This is different from the algebraic approach to specification, where the models (algebras) are implicitly defined by the properties captured in the axioms of the algebraic specifications. A formal Model in VDM is composed of some of following:

- Basic types,
- Defined types (with many useful constructors)
- Invariant of those types,
- Explicit function definition (including precondition)
- Implicit definitions (post conditions)
- Very possibly grouped into abstract data types or classes.

3.4.1 Example of VDM

MSMIE (Multiprocessor Shared-Memory Information Exchange) is a protocol for "inter-processor communications in distributed, microprocessor-based nuclear safety systems" , which has been used in the embedded software of Westinghouse nuclear systems designs. The protocol uses multiple buffering to ensure that no "data-tearing'' occurs as separate processors communicate via some shared memory. In other words, data should never be overwritten by one process while it is still being read by another. One important requirement is that neither writing nor reading processes should have to wait for a buffer to become available; another is that recent information should be passed, via the buffers, from writers to readers. The information exchange is realized by a system with three buffers. At any time, one buffer is available for writing, one for reading, and the third is either between a write and a read and hence contains the most recently written information, or between a read and a write and so is idle [29].

3.4.2 Tools available

SpecBox is an industrialized tool for inputting, checking and printing VDM specifications. It is composed of four main parts: a syntax checker; a LaTeX generator; a semantic analyzer and a translator to the 'Mural' proof assistant. Originally released in 1988, SpecBox is in use in industry in ordnance, avionics, civil nuclear, automotive, security applications and in academic applications. The IFAD VDM-SL toolbox supports syntax checking, extensive static semantic checking, latex pretty printing, test coverage analysis, execution and source-level debugging [30].

.

CHAPTER 4

PROBLEM STATEMENT & OBJECTIVE

4.1. Problem Statement

Use of formal language is increasing day by day. There are many formal languages available, but all of them can't satisfy the requirements for formally specifying all type of problems. Any formal language has its own characteristics as well as limitations. All problems do have its own characteristics. So the problem is how to select a formal language that is suitable for the given problem for specifying it formally in a complete and efficient manner with ease. Various problems are studied to categories them to a certain key characteristic. Then the formal language suitable for specification of that type of problem is to be chosen.

For example, consider the problem of **Local Switch Board (LSB)**- The LSB purpose is to connect user locally, with each other [30]. Local Switch Board is used to illustrate the problems in specification which were not explained by OCL. Here, in case of LSB, UML does not offer means to precisely specify behavior as its state machine semantics is not precisely defined and currently the actions on a transition or describing the body of an operation are specified informally. In OCL we can only specify the signals a class can receive and we can not describe the communication paths, or what happen if two classes can treat a same signal.

4.2 Objective

By implementing the formal methods during the requirement phase in software development lifecycle, we can significantly reduce the cost of fixing error in early stages of SDLC. Since there are a number of formal languages now a day and choosing a formal language for particular problem does not specify all the formal specification completely, if the characteristic of the two do not match. So if a wrong formal language is

chosen for specifying the problem, the result is incomplete specification. For example, in the above mentioned example of LSB (Local Switch Board) if we use OCL for specification, the concept of time has no semantics and it is unclear how it could be actually used. . In OCL we can only specify the signals a class can receive and we can not describe the communication paths, or what happen if two classes can treat a same signal. So need is to use some other formal language.

The primary objective of this thesis is to suggest the formal language which is most suitable in specifying the particular problem formally in a complete and efficient manner with ease. To achieve this following steps have been taken:

- To analyze the case studies of different application areas to identify their key characteristics.
- To compare the characteristics of different formal languages,
- To suggest the best possible formal languages suitable for various types of problems

CHAPTER 5

RESULTS AND ANALYSIS

One of the main goals of software engineering is to provide developers with whatever is necessary for the analysis and design of reliable software, no matter how complex it may be. The use of formal methods (FM) can help to achieve this goal. Formal methods are formal languages, techniques and tools based on mathematics for specifying and verifying computer systems. In addition, the use of FM allows for verification: formal proof that the software does what the specification says it should do. Formal specifications are expressed in a formal language, with well-defined syntax and semantics. There are many examples that show the usefulness of FM in different areas of applications.

Basically the use of term Formal method is to describe any approach which utilizes a formal specification language and specifies the role of that formal specification during the software development process. In other words, the use of formal method will not necessarily imply any formal refinement process, formal reasoning or proof. The term formal specification will then be used to refer to any language in which it is possible to specify fully the functionality of the whole or part of a piece of software in a way amenable to formal reasoning. The emphasis will be on the fact that it is based on a firm mathematical basis, rather than whether it is sufficiently abstract.

5.1 Characterization of Formal languages

Based upon the following properties, we can categorize formal languages as:

Process-oriented – This type of formal language is designed to describe concurrent networks of communicating component's behaviours. Since most of the language describe system in terms of process, so it is called as process-oriented.

Sequential-oriented – It provides the appropriate way to describe the input-output behaviour of sequential system. But one problem with Sequential-oriented is that it lacks concurrence, synchronization and distribution.

Model-oriented – The language which fall in this category provide an abstract model of system and the operations are specified by either state change or by event that affect the model.

Property-oriented – This type of languages focus on the property desired by the system being specified and on data type instead of services/ functions that the system provides.

Formal specification languages are differ from one another by having different properties like-process-oriented or sequential-oriented or it may be a model-oriented or a property-oriented, and what type of mathematics basis it uses in the following way:

Z notation – sequential-oriented, property-oriented, uses the set theory and logics.

VDM – process-oriented, model-oriented, and uses the set theory and logics.

Larch – sequential-oriented, property-oriented, and uses algebra and logics.

Clear – sequential-oriented, property-oriented, and uses algebra.

OBJ – sequential-oriented, property-oriented, and uses algebra and logics.

5.2 Categorization of Problems

1.LSB(LocalSwitchBoard):

The LSB purpose is to connect users locally, with each other. A user interacts with the LSB through his/her phone [31]. The LSB is a simple one, where phone lines and phone numbers are allocated statically at the system startup in a permanent way. Static requirements of LSB supports up to 9 local lines, numbered from 1 to 9. The signals passed from user phones to LSB are: *OffHook, OnHook,* and *DialedDigit*, while the signals passed from LSB to user's phones: *DialTone, BusyTone, RingTone,* and *ConnectTone.* When an action is expected from a user, he/she must react within maximum 15 seconds. The topmost hierarchic entity is the *system*, which can be hierarchically decomposed into intermediate entities called *block*s. A *block* can further contain other *block*s or *process*es (leaf entities). Each *process* contains a state machine that describes its behavior. Entities communicate with each other by exchanging *signals*, which are conveyed by channel that can connect various entities. The LSB system has two blocks one to model the *switch* and one to model the *line handler.* An SDL block is a structural entity that may contain processes (active entities) or other blocks. The two communicate by mutually exchanging messages and the *line handler* also exchanges messages with the *environment*. The *line handler* block contains a process, *line* that describes

the reactive behavior of a line, through a state machine and three procedures: *called, connected, and calling*. The *switch* block describes the actual behaviour of the switch. Concretely, it is composed of *connection agent* that monitors a local call and *connection manager* that receives connection requests from users and allocates connection agents accordingly.

Key Characteristics: Concurrency, Time.

2.ClinicalCyclotronControlSystem(CCCS)

The Clinical Neutron Therapy System at the University of Washington is a cyclotron and radiation therapy facility that provides cancer treatments with fast neutrons, production of medical isotopes, and physics experiments [32]. The control system handles over one thousand input and output signals. The cyclotron control programs are the control subsystem dedicated to assisting the cyclotron operator. Inputs and outputs of this subsystem include a terminal keyboard, two video displays, numerous buttons and lamps on the cyclotron operator's console, and the digital and analog interfaces to the controlled parameters. There are about one hundred and twenty cyclotron control parameters, whose names were assigned by the cyclotron vendor: GASFLOW, DFLVOLT, SWTMAGN, etc. The names for these parameters constitute a Z set:

PNAME = {GASFLOW, DFLVOLT,. . . }

The system state is composed primarily of several quantities associated with each parameter. Most parameters have a quantity called (again following the vendor's nomenclature) pset, which is the control parameter output value most recently written to its digital-to-analog converter (DAC!). The central problem in specifying the cyclotron controls is to define each pset as a function of time, the operators' activities, and potentially hundreds of other quantities, including the input values of other parameters, digital inputs such as interlocks, etc. Z models system behavior as a collection of discrete operations that are also written as schemas i.e. DisplayParam, StatusDisplay, SelectStatus, UpdateStatus etc.

The Z schema composition operator; does provide a built-in way to build up sequential operations from schemas. For our application we would like to define operations that mostly consist of multiple instances of (something like) our TurnOnPS schema running in series and in parallel, with each instance instantiated with different values for its variables. This is another area where other notations may be more suitable than Z. Difficulties arise only when concurrent operations must interact (because they use the same components of the system state). This is a serious practical limitation. Some of the operations described by ChangeParam can take several minutes. It is sometimes necessary for the operator to cancel or modify these operations while they are in progress. For example, the operation SetParam may occur while ChangeParam is in progress, and the new value of psetting determined by SetParam must immediately supercede the final value being approached by ChangePa.ram. There is no way to express this in Z or VDM.

Key Characteristics: Timing Constraint and Concurrency.

3.WallclimbingRobot(WCR)

A wall-climbing robot (WCR) is currently under development at Universiti Teknologi Malaysia. The WCR can be categorized as a small-scale embedded hard real-time system [33].

In hard real-time systems, timing is critical where the lateness of the results is not permitted under any circumstances since late response are either useless or even dangerous. Small-scale embedded hard real-time systems are becoming more sophisticated and usually offer many functions in one product. As a result, software development for small-scale embedded hard real-time systems are growing in scale and becoming very complex over the years. Furthermore, a real-time system is inherently concurrent and multitasking since it has to react to and process numerous events simultaneously. Thus, developing software for even small-scale embedded real-time systems can be very difficult. Due to the complexity

and the nature of the control software for the WCR system, proving the correctness of the software requirements of the robot earlier is important so as to reduce the costs of requirement errors occur in later phases of software development life cycle. The process of building a formal specification of a small-scale embedded hard real-time systems, in particular, the specification of control firmware for a four-legged WCR. The main function of the embedded digital controller is to move the four legs of the robot with a predefined sequence during climbing operation. The main functional operation of the robot controller can be divided into three major groups *i.e.*:

(i) sensors monitoring

(ii) motor control

(iii) serial communication with external PC

The embedded controller monitors its environment using some sensors *i.e.* collision sensors, proximity sensors and pressure sensors. Collision sensors send signal to controller when the sensor collides any obstacles. Obstacle sensors detect the presence of a distance obstacles. This environment must be monitored typically every 500 milliseconds to detect the presence of obstacles using the collision and the obstacle sensors during the forward and reverse movement of the robot. Three position sensors in the form of rotary potentiometers at each leg measure the joint angles of the leg during the leg movement. Position sensors read the current position of the legs joint angle. The highest priority task in the controller software is the motor control task with a cycle of 50 milliseconds. This is to ensure the correct movement of the legs. The set of controller's operational modes is defined by the free type *Modes*. *Yes#o* is the type consisting of the two constants *Yes* and *#o*. *OnOff* is the type consisting of the two constants *On* and *Off*. The following types are used to describe state information of sensors. *Collision_Sensor* denotes the state of a collision sensor; *On* denotes the sensor collides any obstacles, while

Off denotes the sensor does not collide any obstacles. *Obstacle_Sensor* denotes the state of an obstacle sensor; *On* denotes the sensor detects the presence of a distance obstacle, while *Off* denotes the normal state (does not detect the presence of obstacle). *Pressure_Pad* denotes the state of the suction pads of the robot leg; *Yes* denotes enough pressure is maintained in the suction pads, while *#o* denotes the pressure maintained in the suction pads is not enough. *PressureActivate* is used to describe whether an indicator for specific button on a pressure pumps is activated (*active*) or deactivated (*passive*). *Pccommand* denotes the command from the PC to the controller; *Start_Move* denotes the command to the controller to trigger the robot to start moving, while *Stop* denotes the stop moving command. *RobotState* refers to the state of the robot at time t; *Moving* denotes the robot's leg or body is moving, while *Stop_Move* denotes the leg or body is stop from moving; *Idle* denotes the leg or body is not moving. The abstract system's state of the robot controller is specified by the schema *RobotController*.

Key Characteristics: Concurrency, Temporal behavior.

4.ABM(AutomatedBankingMachine)SYSTEM

Automated banking machine (ABM) [34] having a magnetic stripe reader for reading an ABM card, a customer console (keyboard and display) for interaction with the customer, a slot for depositing envelopes, a dispenser for cash, a printer for printing customer receipts, and a key operated switch to allow an operator to restock money in the machine. The stakeholders are the customers, the bank, and the system operator. There are customer related requirements, such as the ABM will serve one customer at a time, operator related requirements, such as the ABM will have a key-operated switch that will allow an operator to start and stop the servicing of customers, and bank related information, such as account number should be 12 numeric-digits. The Z specification of the ABM system (ABMSys) is based on the finite state machine (FSM) representation. The whole

ABM system includes two schema: an ABM machine and an account system. There are four different operations: Withdrawal, Deposit, Transfer, and Inquiry. Each of which will lead to changes in state. Each of the operations defines a relation between before and after versions of the state. The ABM system can perform a transfer if the *machine mode* is *Customer*, the *machine status* is *Busy*, and balance of the "from" account is no less than the amount to be transfered. If the balance of the "from" account is less than the amount to be transfered, an error message, "Error No Enough Fund In Account" is returned. If the *machine mode* is not *Customer* or the *machine status* is not *Busy*, an error message, "Error InvalidOperation" is returned. The Z error schema specifies that the state does not change. The specification for transfer scenario is completed using a schema calculus disjunction:

(Transfer OK) ^ Transfer Error Account) ^ (Transfer Error Operation)

Key Characteristics: Readability, inheritance.

5.ChattaBox

ChattaBox, allowed users to communicate via voice, as well as several other features [35]. The ChattaBox requirements were specified using UML use case diagrams, which specifically allow for engineering of system requirements. These diagrams have three main components: an *actor* (portrayed as a stick figure), a *system* with which the user interacts (portrayed as a box/block) and *use cases* (shown as bubbles inside the system block). Actors can represent human users or systems and their components. In the case of the ChattaBox system, two main requirement groups were identified: *end user* requirements and *internal system* requirements. End user requirements, as stated above, refer to the functionality provided by ChattaBox to a human user. Establishing these requirements involved identifying all the possible desirable functions of the software. The main function that the ChattaBox software had to

provide was to allow users to establish voice calls. Additionally, a number of other feature requirements were identified. The diagram in fig. 5.1 contains a use case diagram that formed a part of the end user requirements specification for ChattaBox.

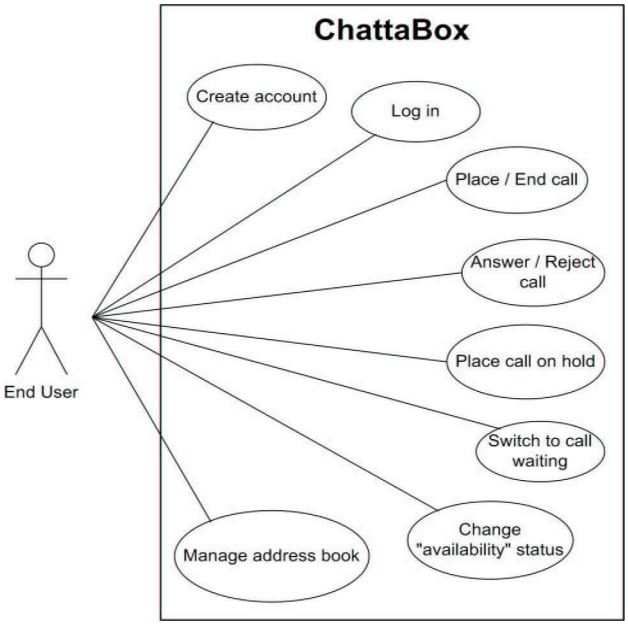

Fig. 5.1: End user requirements for ChattaBox [35]

From the use case diagram it is easy to see what is required of the ChattaBox software from the end user's perspective. ChattaBox was required to keep an account for each user, and provide a logging in facility. Accounts would allow users to keep personalised settings and an address book. Additionally, the software had to allow the user the basic call functionality, such as to place and answer a call. A status service was also to be provided to reflect users' availability to accept calls. Besides the basic functionality mentioned above, advanced requirements were established for ChattaBox, which included voice mail services among others. The ChattaBox server consists of three major components:

1) Remoting Server - This component relies on Microsoft's .NET Remoting infrastructure to make abstracted network calls. It enables

facilities such as user status changes and voice mail. Note that this component relies on the *Security* package for authentication purposes**.**

2) Database - The database component manages persistence of data.

3) SIP Proxy Server - The SIP Proxy server component is responsible for forwarding SIP messages, which are used during session establishment.

The ChattaBox client is composed of four major components:

1) Remoting Client - The Remoting client communicates with the Remoting server, using the Security package for authentication of users.

2) SIP Client - The SIP client sends and receives SIP messages to allow for session establishment.

3) RTP - The RTP component processes real-time voice data, transporting it between two ChattaBox clients.

4) Voice Processor - The Voice processor component captures audio data from the sound input stream to be sent via RTP.

Key Characteristics: Performance, usability, Communication

5.3 Formal language(s) corresponds to type of problem:

Table 5.1: Suitability of formal languages for a particular application area

Type of problem	Formal language(s) suitable	Formal language(s) not suitable & Why
Communication	SDL	OCL **Reason:** It is possible to define a class that can process a set of signal; we have no appropriate way to specify which signals it could send and what consequences are if more than one class can process the same signal. It is even not possible to specify structural containment, in the way as SDL allow it.

Data Modeling , such as Data base system and Purely reactive system i.e. Event driven	SDL	OCL **Reason:** Since the event driven system the behaviour can be described by using State Machine, a good approach for fully specifying their behaviour. So we prefer to use SDL, Because OCL is not very good at capturing communications.
Application having Inheritance and Polymorphism properties	OCL, SDL	Z **Reason:** Since inheritance is one of the charters tics of SDL and OCL basically is the extension of UML which uses the concept if OOP. But in Z, there is no prescribe way for specifying inheritance.
Time interval properties type application	SDL	Z, VDM **Reason:** They do not provide any build-in way to express the passage of time.
Concurrency	SDL, Petri Nets.	Z, VDM **Reason:** They do not provide any build-in way to express concurrency. However, they do not preclude describing concurrent activities. Difficulties arise only when concurrent operations must interact. (ClinicalCyclotron Control System)
Predefined Data System	SDL	OCL **Reason:** Since OCL is just allowing some constraint on UML class diagram and UML has no data system. Also there is a lack of basic data type in data packages and provides no support for collection definition or other type of data definition whereas SDL support a whole data type and data type definition mechanism.

Pure mathematical problem(which uses more mathematical symbols and functions)	Z	SDL,VDM **Reason:** Since SDL uses only abstract data type and use process and block for specifications. There is no in-built data type for partial functions in SDL and VDM.

CHAPTER 6

CONCLUSION & FUTURE SCOPE

After analyzing the characteristics of formal languages and problems, formal languages which are most suitable in different scenarios are specified. This will help in choosing the right formal language for complete specification of a given problem.

Characteristics identified are:

SDL is suitable for problem involving communication, inheritance, concurrency and real-time related problems.

OCL is suitable for problem involving inheritance, more readability, and reusability.

Z is suitable for problems involving more mathematical symbols like partial functions, domain & range of a function, small embedded real-time application.

VDM is suitable for security applications, mathematical properties like mapping etc.

In future, the applicability of more formal languages (i.e. LARCH, HOL, and B-Method etc) can be studied.

References

[1]. M. Osborne and C.K. MacNish, "Processing Natural Language Software Requirement Specifications", Proceedings of the Second International Conference on Requirements Engineering, 15-18 April 1996, pp: 229 - 236

[2]. K. Johannisson, "Formal and Informal Specifications", PhD Thesis, Chalmers University of Technology, 2005

[3]. L. Lamport, "An axiomatic semantics of concurrent programming languages". In K. Apt, editor, Proceedings NATO Advanced Course on Logics and Models of Con-current Systems, pages 77–122. Springer-Verlag, 1985.

[4]. J.A. Serrano, "Formal Specifications of Software Design Methods", 3rd Irish Workshop on Formal Methods Galway, Ireland. 1-2 July 1999

[5]. "Software Requirements Engineering", SEI Interactive, Mar'99 9 (http://www.sei.cmu.edu/interactive/Features/1999/March/Introduction/intro_ mar99.htm)

[6]. A.P. Eberlein, "Requirements Acquisition and Specification for Telecommunication Services", PhD Thesis, University of Wales, Swansea, UK, November 1997

[7]. N.W. Morgan and C. Schahczenski, "Transitioning to Rigorous Software Specification", Proceedings of the First International Conference on Requirements Engineering, 18-22 April 1994, pp:110 - 117

[8]. B.L. Charlier and P. Flener, "Specifications are Necessarily Informal or: Some More Myths of Formal Methods", Journal of Systems and Software, 1998, pp:275 – 296

[9]. R. Fernandes, A.J. Cowie, "Capturing Informal Requirements as Formal Models", 9th Australian Workshop on Requirements Engineering, Adelaide, South Australia, 2004

[10]. R.S. Pressman, "Software Engineering – A Practitioner's Approach", 5Ed, McGraw Hill, 2001, 0-07-118458-9

[11]. F. Fabbrini, M. Fusani, V. Gervasi, S. Gnesi and S. Ruggieri, "On Linguistic Quality of Natural Language Requirements" In 4th International Workshop on Requirements Engineering: Foundations of Software Quality REFSQ' 98, Pisa, June 1998.

[12]. Shaoying Lui, "Formal engineering methods in software devlopment" Department of computer and Information Sciences, Hosei University.

[13]. D.O. Paun and M. Chechik, "On Closure under Stuttering", Formal Aspects of Computing, Volume 14 , Issue 4, Springer-Verlag, April 2003, pp: 342 -368

[14]. C. Denger, D.M. Berry and E. Kamsties, "Higher Quality Requirements Specifications through Natural Language Patterns", Proceedings of the IEEE International Conference on Software-Science, Technology & Engineering (SwSTE'03), 4-5 Nov. 2003, pp:80 – 90

[15]. N.E. Fuchs, "Specifications Are (Preferably) Executable", Software Engineering Journal, vol. 7, no. 5, September 1992, pp: 323 – 334

[16]. M. Satpathy, C. Snook, R. Harrison and M. Butler, "A Comparative Study of Formal and Informal Specifications through an Industrial Case Study", IEEE/IFIP Joint Workshop on Formal Specifications of Computer-Based Systems, Washington DC, April 2001, pp. 133-137

[17]. F. Fabbrini, M. Fusani, S. Gnes and, G. Lami, "An Automatic Quality Evaluation for Natural Language Requirements", Proceedings of the Seventh International Workshop on RE: Foundation for Software Quality, 2001

[18]. A. Gravell, "What is a Good Formal Specification", December 1990 Proceedings of Proc. 5th Int. Conf. Annual Z User Meeting, pp. 137-150

[19]. J. Bowen, "Formal Specification and Documentation using Z: A Case Study Approach", Revised Edition, 2003 (http://www.afm.sbu.ac.uk/zbook)

[20] J.P. Bowen and M.G. Hinchey, "Seven More Myths of Formal Methods", IEEE Software, July 1995, pp. 34-41

[21] [Spi92] J.M. Spivey: " The Z Notation- a Reference Manual", 2nd edition. Prentice Hall, 1992.

[22] Tool support for Z, "http://academic.uofs.edu/faculty/beidler/zed/"

[23] Warmer, J., and A. Kleppe " Object Constraint Language", edition-Wesley 1998.

[24] Tool support for OCL "http://www.klasse.nl/ocl/ocl-tools.html"

[25] Zhang Yaoxue, Chen Hua, Zhang Yue and Liu Guoli, "SDL-Tran- An Interactive Generator for Formal Description Language SDL" , Department of Computer Science & Technology, tsinghua University, Bejing, 1993.

[26].Tool support for SDL "http://www.sdl-forum.org/Tools/Commercial.htm"

[27] C.B. Jones, " Systematic Software Development using VDM", 2nd ed. Englewood Cliffs, NJ: Prentice- Hall, 1990.

[28] Pedersen, J.S. and Klein, M.H. " Using the Vienna Development Method (VDM)" Tech. Rept. CMUSEI-88-TR-26, SEI, Software Engineering Institute, Carnegie Mellon University, Nov. 1998.

[29]. Savi Mahraj, Juan Bicarregrei, "On the Verification of VDM Specification and Refinement with PVS"

[30] Tool support for VDM http://www.vienna.cc/e/evdm.htm

[31] Philippe Leblanc, Ileana Ober, "Comparative Case Study in SDL and UML".

[32] Jonathan Jacky, " Formal Specification for a Clinical Cyclotron Control System".

[33] Radziah Mohamad, Dyg. Norhayati ABG. Jawawi, Safaai Deris and Rosbi Mamat,"Formal Specification of a Wall-Climbing Robot using Z- A Case Study of Small- Scale Embedded Hard Real- Time System"

[34] Munina Yusufu, Gulina Yusufu, "Comparison of Software Specification Methods using a Case Study".

[35] P S Kritzinger, M Chetty, J Landman, M Marconi and Ryndina, " ChattaBox: A Case Study in Using UML and SDL for Engeineering Concurrent Communicating Software Systems".

List of Publications

PUBLISHED

- Monika Dhariwal and Ms. Shivani Goel, "*Comparison of Formal Methods based upon Various Parameters",* International Advance Computing Conference (IACC'09), Thapar University, Patiala, India (6-7 March, 2009).

COMMU ICATED

- Monika Dhariwal and Ms. Shivani Goel, " *Identifying the Suitability of Any Formal Language for a Particular Application Area*", International Journal of Information Technology and Knowledge Management (IJITKM) , Kurukshetra University, Kurukshetra, India.