Converting Natural Specifications to Z

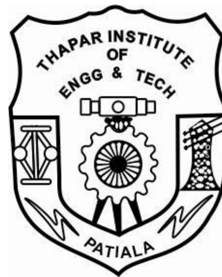
A Thesis

submitted in partial fulfilment of the requirements

for the award of degree

of

Master of Engineering
in
Software Engineering



By:
Rocky Goyal
(8043120)

Under the supervision of:
Mr. Rajesh K. Bhatia
(Assistant Professor)

MAY 2006

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY
(DEEMED UNIVERSITY)
PATIALA – 147004

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Converting Natural Specifications to Z",** in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering at Computer Science & Engineering Department of Thapar Institute of Engineering & Technology (Deemed University), Patiala, is an authentic record of my own work carried out under the supervision and guidance of Mr. Rajesh Kumar Bhatia.

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

(Rocky Goyal)

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

(Mr. Rajesh Kumar Bhatia)
Assistant Professor
Computer Science & Engineering Department
Thapar Institute of Engineering & Technology
Patiala- 147004

Countersigned By:

(Dr. (Mrs.) Seema Bawa)
Professor and Head
Computer Science and Engineering Department
Thapar Institute of Engg. and Tech.
Patiala

(Dr. T.P. Singh)
Dean
Academic Affairs
Thapar Institute of Engg. and Tech.
Patiala

Acknowledgements

The works like this are never completed single handed. There are many a persons working behind the screen but are equally important. I take this opportunity to express our deep sense of gratitude towards all those.

First of all, humble bow before the almighty God for his blessings and providing the power to finish the work.

I thank the computer science and engineering department of Thapar Institute of Engineering and Technology for giving me opportunity and providing resources for carrying out the research and completing thesis work.

I take the opportunity to thank my supervisor Mr. Rajesh K. Bhatia, Assistant Professor, who has been a continuous source of inspiration for me, for his encouragement, patience and providing invaluable guidance throughout this work, especially when I was unsure about which direction to take.

From a personal side, I would like to thank my colleagues and friends, without whose support, I might not be able to finish the thesis work. I thank Kapil Bhatia for reviewing the first drafts of the report. My thanks are also due to Rajiv Bammi for his valuable advice and moral support during the thesis.

Last but definitely not least, I would like to thank my parents and family members, for their support throughout the work.

Rocky Goyal

Abstract

Formal specification of systems has been an active area of research for quiet some time. In software engineering, the formal specification of the requirement phase is of utmost importance to achieve rigorous development and maintenance of software systems. Despite extensive development over many years and significant demonstrated benefits, formal methods remain poorly accepted by industrial practitioners and natural language remains the first and practical choice of software engineers for specifying the system. The main reason behind the use of natural language for specification writing is the ease of use and power of expression, whereas formal methods are difficult to learn and master. This lack in ease of use is a hindrance to the use of formal methods to their full potential. But the use of formal methods is of utmost importance in safety critical systems. The formal specifications lead to development of high quality software systems. There is a large gap between the actual capability of the formal methods and the purposes for what they are presently being used

This thesis presents an approach for conversion of software specifications written in natural language to a formal notation using Z. This will help in exploiting the strength of formal methods, such as their use in software reuse, verification and validation, while keeping intact the ease of use of natural language in specifications. The thesis gives an overview of natural language specifications and formal methods, discussing their respective benefits and drawbacks. Then a system for conversion of natural language specifications to Z is proposed.

Contents

Certificate i

Acknowledgements ii

Abstract iii

Contents iv

List of Figures vi

Chapter 1 Introduction 1

1.1 Motivation and Objective 2

1.2 Organization of Thesis 4

Chapter 2 Background Information 6

2.1 Requirement Engineering 6

2.2 Natural Language Specifications 12

2.3 Formal Methods 15

2.4 Natural Language Processing (NLP) 24

Chapter 3 Literature Review 28

3.1 Natural Language Representation of Formal Specification Models 28

3.2 Converting Natural Language Specification to Formal Notation 30

Chapter 4 Proposed System and Implementation 37

4.1 Problem Statement 37

4.2 Why is it worthwhile to address the problem 38

4.3 Overview of the System 38

4.4 Working of the System 39

4.5 Example System 40

4.6 Features of System 41

Chapter 5 Conclusions and Future Work 45
5.1 Conclusions 45
5.2 Future Work 46
References 47
Appendices 54
Appendix A: Glossary of Z Notation 54
Appendix B: POS Tags 59
Papers / Publications 63

List of Figures

Figure 2.1: Software Development Costs 20
Figure 2.2: Generic Structure of Schema in Z 23
Figure 2.3: The NLP Process 25
Figure 3.1: Architecture of REVIEW system 28
Figure 3.2: Translating OCL to Natural Language 29
Figure 3.3: The Borg Tool Overview 31
Figure 3.4: Schematic View of SPECIFIER system 33
Figure 3.5: System Structure [62] 35
Figure 3.6: The conversion process [8] 36
Figure 4.1: The architecture of the FREND system 39
Figure 4.2: Working of FREND 39
Figure 4.3: Example - Formal Specifications in Z 41
Figure 4.4: Partial View of AddRule Screen 43
Figure 4.5: Editing/Adding a Schema 43
Figure 4.6: No schema found Screen 44

Chapter 1

Introduction

For the development of software, the starting point is usually a set of requirements, known as system specifications. These specifications are written in a document known as System Requirement Specifications (SRS) [1, 2, 3]. The SRS often form the basis of the contract between the customer and the system supplier [1, 4, 5, 6]. Consequently, they need to be precise so that they can be taken as a basis of a specification of what the system is supposed to do. They also need to be clear enough for customers to confirm that a system built from the SRS satisfies their needs. Thus the system specifications need to be clear and precise. These features of the specification document depend on the particular requirement specification technique used to create the SRS. For example, SRSs expressed solely in natural language may be comprehensible by the customer, but an SRS expressed in this way will be ambiguous, possibly inconsistent. SRSs expressed in terms of formal techniques may be clear to the system designers, but will be difficult to comprehend by non-computer specialists [1].

The software specifications are generally described using a natural language like English [3, 7]. In software engineering, the formal specification of the requirement phase is of utmost importance to achieve rigorous development and maintenance of software systems [8]. A number of formal specification methods are available. Even today, the formal methods of software specification are not very popular among the software professionals and still the natural language remains as the first and practical choice of the experts to specify the system [9, 10, 11, 12, 13]. The main reason behind this may be the hardship to master the formal specification languages and their inappropriateness in communicating the requirements [9, 14, 15]

The software specifications are generally written in natural language and then converted manually to formal methods [16]. The manual process becomes complicated and thus is prone to errors, especially when the system to be specified is large in size. The automated conversion or the computer aided conversion of natural

language specifications to formal language can reduce the complexity of the manual translation process even for specifying the complex and large systems.

Although the system specifications can directly be written in a formal language, natural language description of the system is indispensable. Even at the later stages of software life cycle, design and implementation, the natural language description is used, for example comments in writing programs [17]. However, there always is a question on the adequacy of informal content added to a formal specification [18]. Nevertheless, the two approaches for system specification i.e. formal and informal are complementary rather than competing [19].

In this thesis, an approach to automate the task of conversion of system specifications written in natural language to a formal language notation is presented. The natural language used is English, whereas Z acts as a surrogate for the formal method of software specification.

1.1 Motivation and Objective

The main objective of the thesis is to identify and show the usage of natural language processing in software engineering. As most of the natural language usage is in the initial phases of software lifecycle, i.e. analysis and requirements phase, these phases were particularly investigated for the possible use of NLP techniques. On investigating, I found that the inherent problems in natural language description of a system can cause errors to creep in. The formal methods seemed to be promising in reducing the chances of errors, and hence producing the high quality software systems. Despite extensive development over many years and significant demonstrated benefits, formal methods remain poorly accepted by industrial practitioners [20]. Keeping all these things in mind, the review of the following domains was done to meet the objective, and phrase the problem for the thesis

- Natural Language Processing
- Requirements Engineering
- Natural Language Specifications
- Formal Methods

Natural language processing is about making computers able to understand and interpret the text written by human beings. This aims at making interacting with

computers using natural language as easy and natural as communicating with other human beings.

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

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 standards [21]. Formal methods are still not used to the full of their capability due to the lack of ease of use of formal methods [22].

More than half of the errors that appear during the entire system's development lifecycle, originate at the requirements phase. The cost and effort required to correct an error detected at later stages is manifold than those found and fixed at the requirements stage [23, 4, 24, 6, 25, 26, 20, 27]. It is useful to capture the specifications in a formal notation rather than the informal specifications written in natural language. The advantage of formal specification is that it can be checked for consistency, and completeness to some extent, by using the tools available [4]. The informal specifications on the other hand are easily understood by the user. There has to be synchronisation between the two specification methods so that the changes made in one are automatically reflected in the other [14], because no single language, either formal or informal, fits all the needs [18].

In this thesis, an attempt to bridge this existing gap in formal and informal specification methods is done. This thesis aims to provide a framework for the effective use of formal methods in the early phase of requirements engineering. The motivation of the thesis is to make a link between the informal specifications used as a practice in software engineering and the formal specifications that are required to exploit the formal methods. In the thesis, the obstacles in the use of formal methods are identified and some of them are tried to be removed, thus encouraging use of formal specification methods to make use of their known advantages. Another motivating factor for the thesis is the amount of reduction of rework required when

the formal methods are used early in the development lifecycle, right from the requirements phase [24].

The objectives of the research include

- To show the feasibility of converting natural language specification of a system to the formal notation
- Facilitating the conversion of natural language specifications to formal specifications, thus bridging the gap
- Enabling the people who are not well versed with formal methods, to use formal methods to some extent and hence benefit from them, by providing a conversion framework from natural language to formal specifications
- Development of a tool using NLP, to automate conversion of natural language specifications to formal ones
- Facilitating the use of natural language documents for requirements engineering purposes and communicating with non-technical users, and the use of formal methods for modelling, implementation and verification of the system.

The thesis however, neither claims to nor aims at removing the human cognition aspect from the requirement engineering activities, including the conversion from natural language description to formal notation.

1.2 Organization of Thesis

The rest of the thesis is divided into 4 chapters. Chapter 2 gives an introduction to the various domains described in section 1.2. The information presented in chapter 2 acts as a prerequisite for understanding of rest of the thesis. The chapter is divided into 4 sections, each addressing a particular domain. 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 requirements specified in natural language or descriptive notation is given. The section also takes a look at the pros and cons of using natural language as an informal method of requirement specification.

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 respective benefits and problems in their implementation in real systems are discussed. The section also gives an overview of some of the existing formal methods, with Z formal method covered in relatively detailed manner.

Section 2.4 is a review of NLP techniques and describes how it is used in the presented thesis.

In chapter 3, the research work in this and related domains is discussed. It introduces various techniques being used for making link and bridging the gap between formal and informal software specifications.

Chapter 4 discusses the problem in detail and the solution suggested thereof. It details the actual implementation of the system, giving technical information about the system and also acting as a guide to the user of the program. The use of system is also elaborated in the chapter by taking a small example of a simple library system.

Chapter 5 reviews and concludes the thesis work. Directions and guidelines for the future work are also discussed in this chapter.

There are two appendices at the end of this thesis. Appendix A describes the mathematical notation used for Z specifications and appendix B summarises the tags and their corresponding parts of speech as used in the thesis and the system developed.

Chapter 2

Background Information

This chapter gives an introduction to various domains that have been investigated for the completion of this thesis. 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 requirements specified in natural language or descriptive notation is given. The section also takes a look at the pros and cons of using natural language as an informal method of requirement specification.

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 respective benefits and problems in their implementation in real systems are discussed. The section also gives an overview of some of the existing formal methods, with Z formal method covered in relatively detailed manner.Section 2.4 is a review of NLP techniques and describes how it is used in the presented thesis.

2.1 Requirement Engineering

"Good begun is half done"

Requirements describe the desired functionality of a system. In general, there are two types of requirements: functional and non-functional. Functional requirements describe the behavioural aspects of a system; non-functional requirements describe the non-behavioural aspects of a system. Requirement Engineering is the initial and a critical phase of system development life cycle [28, 29, 30]. It has become a key issue as requirement engineering is responsible for maintaining the requirements of a system over time [30]. The requirement engineering process influences the overall success of the project [28, 31]. The requirements phase is an important phase of a project. Even a few inaccuracies or misunderstandings in requirements can increase

the overall development and test effort. Requirement errors found late in the development cycle can create weeks of re-work by developers and testers [36].

Requirements specification should precede system design. Specifications may be influenced by major design decisions but for the main part they should be independent of implementation details. Detailed data flow diagrams and pseudo code should not be part of a requirements definition or a requirements specification document [36].

As customer satisfaction is becoming the predominant measurement of a system's quality, correctly understanding, documenting and validating the needs of the stakeholders involved in the development becomes more and more crucial. The correctness of product requirements is the key point determining the success or failure of project development [37]. Some researchers and developers believe that the requirements should only state what a system should do rather than how it should do it [30, 38].

A program specification is a statement which describes

- what purpose the program serves and
- how the program can be correctly used [32]

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 [33]. 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. As also discussed in chapter 1, the cost of detecting and correcting errors at later stages of development cycle increases exponentially [30]. 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 [24, 34]

- **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" [30]. 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 [35]. It also requires a good deal of customer contact and domain knowledge [34]. Identification of the stakeholders is also an essential task in requirements elicitation [30]. The requirements elicitation also includes context analysis, which addresses the purpose of the system – why the system is being developed at all [30]. 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 categorisation of requirements and their organisation 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 [34]. 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 [30].

- **Requirement Specification:** Requirement specification aims at the production of an SRS. The main purpose of a requirements document is to convey information gathered from the customer and other sources to the developer [39]. A specification can be a written document, graphical model, a formal or mathematical model, mental model or a combination of

these [30, 34]. Requirement specification languages can be categorised into following classes:

- **Informal languages:** Text and recordings in natural language, as well as pictures and animations, fall into this category. They are very expressive. “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, contradictions, incompleteness and misunderstandings [30]. Section 2.2 describes the natural language as a method of requirements specification in detail. These are good for specifying large software systems.
- **Semi-formal languages:** Entity relationship diagrams (ERD), data flow diagrams (DFD) and state transition diagrams (STD) 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 [30].
- **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. The formal methods are discussed in section 2.3.

The selection of a category of specification languages and a language thereof 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 criticalness 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 [36].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 [34]. 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 [30]. 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. Such a list of questions is given in [34].
- **Requirements Management:** Requirement management is the set of activities carried out to identify, control and track requirements and changes to requirements [34]. 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 [30]. 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 [30]. The main reason for the need of requirement 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:
  - **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.

  - **Change in External Interfaces:** The requirements may be changed, added or 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.
  - **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.
  - **Requirements-sources traceability:** Links the requirement with the people or documents that specified it.
  - **Requirements-rationale traceability:** Links the requirement with a description of the rationale for it. It is concerned with the sensibility of the requirement specified.
  - **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 [34]
  - **Requirements-architecture traceability:** Links the requirement with the sub-system in which it is implemented. This is especially

important for sub-contracting. It can also said to be subsystem traceability.

- **Requirements-design traceability:** Links the requirement with specific design components that are used to implement the requirement.
- **Requirements-interface traceability:** Links the requirement 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 re 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 Natural Language Specifications

For a software project to be successful, it is essential that the requirements of a software system reflect the user's needs [40]. The software specifications are generally described using a natural language like English. Natural language is convenient because it allows non-technical users to understand the product requirements [24]. People are more comfortable explaining things in the language they use in their daily life. This applies to explanation of software requirements as well. Most people find it easier to express specifications in a natural language. Ease of use and freedom of expression is the key reason for its popularity. Informal specifications can offer advantages in readability and better understanding [4]. The SRS serves as a contract between the customer and the developer. So it is important for the customer to know what has been described in the document. The customer is one of the users of the SRS document, along with other technical users. Using natural language for such a document is preferred and advisable because it allows non-technical users to understand the product requirements [24].

There are problems that creep in when natural language is used as a medium for description of the software. When using natural language, ambiguities and

contradictions are unavoidable side effects [4]. The lack of precise semantics increases the possibility of errors being introduced due to interpretation mistakes and inherent ambiguity [24, 37, 40, 41, 42, 43]. Under or over specification are also common problems when using natural language [24]. Ambiguity in requirement specifications can cause numerous problems in defining customer/supplier contracts, ensuring the integrity of safety-critical systems, and analysing the implications of system change requests [1].

Restricted forms of natural language are widely used as a notation for software requirements specifications [24]. Such a form of natural language is called controlled natural language or simply controlled language (CL). A controlled language is usually a subset of a natural language, which is constrained in its lexicon, its grammar or its style, and which may be extended with domain specific terminology and expressions. When designing a controlled language the usual goal is to develop a language which is less ambiguous than natural language, since this will make text manipulation by humans or machines easier. For example, a technical document written in English that is less ambiguous than Standard English and using an agreed technical vocabulary will be easier to translate into different languages by humans or by machines. Sometimes, it is possible that the texts may become so unambiguous and easy to understand that translation is no longer necessary [44].

For example Berry and Kamsties [45] state that there should be careful use of plurals in SRS. They state that the statement

"All the lights in any room have a single on-off switch."

is ambiguous in nature, because it has two interpretations viz.

1. All the lights in any given room share a single on-off switch.
2. Each light in any room has its own on-off switch

They recommend using 'each' for addressing every single entity in a set, and 'all' when talking about the set as a whole.

Restricted forms of natural language are widely used as a notation for software requirements specifications [24]. Some of the well known controlled languages are the following [30]:

- **AECMA Simplified English:** AECMA Simplified English is the best known of controlled languages. It is used, and in fact is mandatory, in many aircraft maintenance manuals. The goal of AECMA Simplified English is to make these manuals easier to understand by non-native speakers, since the manuals do not get translated.
- **Caterpillar Fundamental English:** This language is used by Caterpillar, agriculture and heavy machinery manufacturing company, for their maintenance manuals. These manuals need to be translated into the main languages of all countries where Caterpillar sells equipment. The goal of the controlled language is to make the manuals less ambiguous and more predictable. This reduces the cost of human and machine-assisted translation, either because translations can be reused or because translators spend less time trying to understand ambiguities in the source text.
- **PACE:** Perkins Approved Clear English was designed by technical writers at Perkins and is used for their maintenance manuals. The original goal was to use the controlled language to ensure that all manuals adhere to a particular house style, so that manuals can be written by different authors and partly reused at different times without causing stylistic ruptures. More recently it has also been used to cut down on the cost of translation.
- **ACE**: Attempto Controlled English ACE is a research prototype for a controlled language, designed to be used by experts in software specification. The goal is that input text will be mapped unambiguously into first-order logic (via an intermediate mapping into Discourse Representation Theory (DRT), and the user is presented with a set of principles and recommendations which constrain the lexicon and syntax of the input specification.
- **CPE:** It is the abbreviation used for Computer Processable English. The previous languages were all designed by taking as a starting point an intuitive notion of what constitutes a less ambiguous, easier to understand language. Processing tools are then developed which can deal with this language. CPE was designed differently: if the goal is to design a controlled language which will allow certain text processing tasks to be

automated, then why not start from the capabilities of an existing text processing tool and see whether that gives you a reasonable approximation of a controlled language. SRI Cambridge tried this, using the coverage of their CLE language processing environment as a starting point to define a controlled language. Further constraints were added to reduce the cost of resolving anaphoric expressions. It has so far only been used as a research prototype.

It is not always possible to use controlled natural language for the following reasons:

- Controlling a natural language reduces the habitability of the system. That is, the CL might be so restricted that it becomes irritating to use and arguably becomes a formal, unnatural language. For such cases, the user might be better off actually using a formal language. However, trying to reduce the gap between the CL and unrestricted natural language re-introduces the above problems [1].
- The user needs guidance on how to phrase requirements in terms of the CL. This means that when the user deviates from the CL, appropriate information is given, enabling linguistically naive users to re-phrase their statement in terms of the CL. Also, it is often difficult for humans, especially the writer or the specifier himself, to determine why a sentence is ambiguous, and what the alternative readings mean. Hence, the system needs to present the alternatives clearly [1].
- Natural languages are not always an appropriate medium for expressing all requirements. For example, algorithms are better expressed in some sort of task-related formalism; structural relationships are better expressed in terms of diagrams, and so on [1].
- The use of a CL can also result in reduced power of expression; reduced writing speed; and the difficulty of obtaining compliance from users [44].

2.3 Formal Methods

The Encyclopaedia 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" [34].

Formal specification of systems has been an active area of research for quiet some time. [46]. 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 [40]. 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 are rigorous methods used in system design and development. They use mathematics and logic to prove system correctness, and are used to increase confidence in the system. Formal methods have been advocated as one way to define requirements with mathematical precision and rigour [24]. Traditionally, specifications have been written in natural language, but today more and more specifications are written in formal specification languages [48]. They are used regularly while verifying systems that are of critical importance or real time event driven, where errors are not acceptable [14, 49].

Anyone involved in any stage of system development can make use of formal methods. They can be used in the initial statement of a customer's requirements, through system design, implementation, testing, debugging, maintenance, verification, and evaluation. Formal methods are used to reveal ambiguity, incompleteness, and inconsistency in a system. When used early in the system development process, they can reveal design flaws that otherwise might be discovered only during costly testing and debugging phases. When used later, they can help determine the correctness of a system implementation and the equivalence of different implementations [33].

However, formal methods just specify what the system is to do and not how it must be done [47]. A formal specification is a statement in a formal specification

language [32] Formal specification of requirements phase is very important in achieving rigorous development and maintenance of software systems [8]. 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 [50]. They provide the means to build system models by specifying their structure and behaviour. When used in conjunction with tools support, automated verification of properties for complex models is also possible [24].

Formal specifications have the additional advantage over informal ones of being amenable to machine analysis and manipulation [33]. 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 is the only way to be precise about a system before it is built.

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 [36]

Formal methods are convenient for specifying large software systems, where completeness and consistency are important issues [41].

The formal methods can be used

- To clarify your understanding of the problem
- To communicate your intentions to other developers
- To provide a prototype to demonstrate your 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[53]

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

- Seek commitment from all levels: management to system engineers.

- 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.
- 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.
- Start with a small project. Early success on the small project will help gain buy-in for the formal method process.
- Spend time deciding what should, and what should not, be formally specified. Specify only those portions of the system that need clarification.
- 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. [36]

In addition to these recommendations, there are 'The Ten Commandments of Formal Methods' as described in [34], [56].

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 quality 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 that are discussed in [56] and [57]

- **Myth 1: Formal Methods can guarantee that software is perfect:** Any technique is fallible, and formal methods is 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 be 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 of 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 mathematic 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.

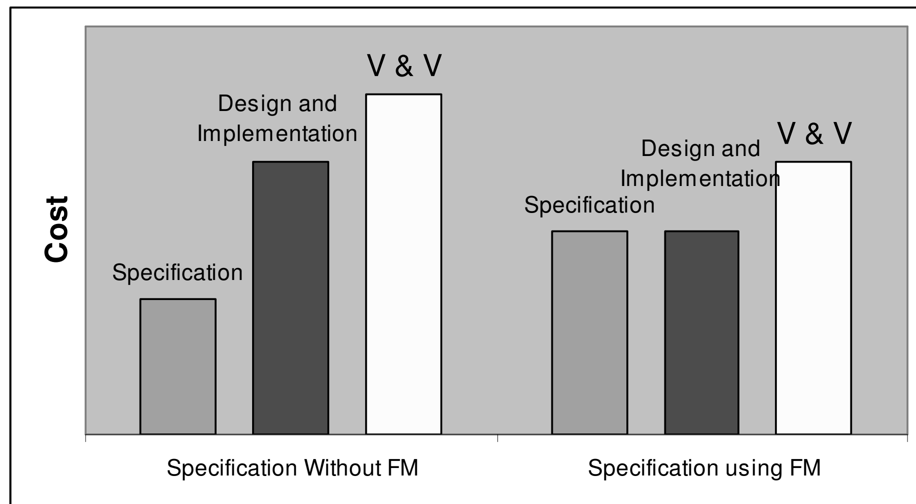


Figure 2.1: Software Development Costs

- **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.

A variety of formal specification languages are available. Following paragraphs give a brief overview of the formal specification languages commonly in use. The Z specification language is discussed in a more detailed manner.

- **HOL:** HOL is an interactive theorem-proving system based on Higher Order Logic, used for constructing formal specifications and proofs. It has been applied in industry and academia to support formal reasoning in hardware, real-time systems, compiler verification, etc.

- **LARCH:** LARCH is a specification language supporting equational 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.

- **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.

- **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.

- **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.

- **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. 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. [34][58].

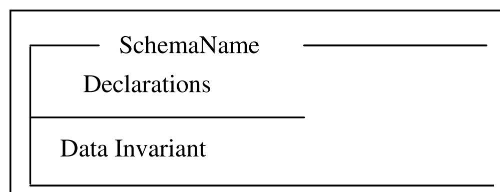


Figure 2.2: Generic Structure of Schema in Z

The notation is useful, once it has been learned, to organize the thoughts and aid the communication of ideas within a design team. It is also readable enough to be used as a documentation tool in a manual. The

mathematical notations and conventions used in Z are described in appendix A.

Although formal methods have been applied successfully in industry and research projects, they are still not widely used today. The reasons for formal methods not being popular are discussed in the following text.

Being based on mathematical frameworks and notations, they are not as intuitive to the general audience as natural language [24].

The disadvantage of traditional formal languages is that they are useable to persons with a string mathematical background, but difficult for the average business or system modeller to use [51]. Formal methods are difficult for most human readers to understand [41].

The specifications can be positive or negative. A Positive specification dictates that programs must do something. A negative specification dictates that programs must not do something [52]. It is possible to write positive specifications in formal way, but not negative specifications.

A significant amount of effort can be required to learn a formal specification language [54] At the same time, there is increasing pressure to minimize the cost and schedule time for software development efforts, giving software developers even less time to learn effective use of formal techniques [54]. Many domain specialists are not familiar or comfortable with formal notations and formal tools [53]. Though formal methods promise improved quality of software and partial automation of the software development process they are not readily accepted by domain specialists [53]. As a consequence, the great majority of software requirements continue to be specified in natural language [54]. In the next section, we give an overview of natural language processing.

2.4 Natural Language Processing (NLP)

Natural Language Processing (NLP) is to make computers able to communicate with human beings using natural language. Since the development of computers, attempts are being made to make computers understand and use natural language. When we, as humans see or read text, we understand its meaning; but computers when see text, they interpret only words and lines. Thus the main objective of the

NLP is to make computers understand the natural language. Rich and Knight define understanding as “the process of mapping from an input into a more immediately useful form” [59]. NLP is mapping the natural language to machine language and vice versa to make the communication between the man and the machine more effective and easier.

NLP can be divided into computational linguistics and theoretical linguistics. The main aim of **Computational Linguistics** is to develop systems for processing natural language. It also aims to handle mist cases of natural language and can cope with approximations and inexact solutions. They do not care about occasional failures and are more concerned with getting the system work. It is an engineering discipline rather than scientific one. The input for computational linguistics is theoretical linguistics. The primary concern of the **Theoretical Linguistics** is things like grammatical coverage, principles of grammar, grammatical formalisms, determining the single universal and ultimate grammar etc.

As stated earlier, natural language processing involves mapping of natural language into a computer language and then retrieving information by converting it from computer understandable language to a natural language. The underlying process is shown in the figure below.

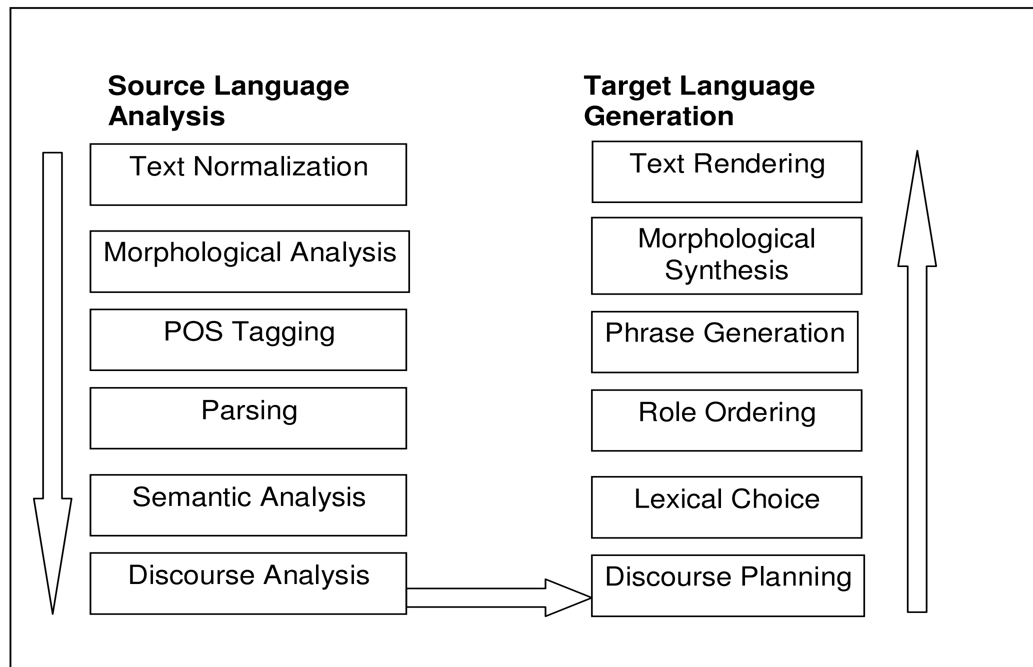


Figure 2.3: The NLP Process

Each phase of the NLP process in described briefly below.

- **Morphological Analysis:** Morphology is the study of form and structure of word formation. In word formation we study about the stem of the word, its affixes i.e. the prefix and the suffix. Affixes are used to alter the meaning of a word or to make new words. In morphological analysis, the words are separated from non-word tokens, such as punctuation marks. This is thus also referred to as tokenization. Actually the morphological analysis is divided into two parts viz. tokenization and tagging. Tokenization is recognizing and separating the individual words.

- **POS Tagging:** Tagging is the process of attaching the parts of speech (POS) tags to each of the words. Each word is associated with some features or properties. E.g. dog is a noun and is a singular. It is useful because the interpretation of the affixes may depend on the syntactic category of the word. The words may have more than one syntactic category. For example, the word 'needs' can be used as a plural (of need) noun or a third person singular verb (as in "He needs"). The tagging algorithms attempt to assign most likely tag for the given word. Probabilistic models are used for tagging. Most of the statistical algorithms are counting based and find the Maximum Likelihood Estimate (MLE) to get the most probable tag and assign it to the particular word. Hidden Markov Model (HMM) is one such model for tagging. This POS tagging is used as a basis in the given thesis for understanding the natural language constructs and their conversion to a formal notation. Brill's tagger based on Brown Corpus is used in the development of the system. The Brill's tagger works as follows to assign tag to a word.

  Tagging is done in two stages. Every word is assigned its most likely tag in isolation. Each word in the tagged training corpus has a lexical entry consisting of a partially ordered list of tags, indicating the most likely tag for that word, as well as all other tags seen with that word (in no particular order). A list of transformations is provided for determining the most likely tag for words not in the lexicon. Unknown

words are first assumed to be nouns (proper nouns if capitalized), and then cues based upon prefixes, suffixes, infixes, and adjacent word co-occurrence are used to change the guess of most likely tag. Next, contextual transformations are used to improve accuracy.

A detailed list of parts-of-speech and the corresponding tags generated by Brill's tagger is given in appendix B.

- **Syntactic Analysis:** This process is also known as parsing. Once the building blocks of a sentence have been determined to be correct (morphological analysis), syntax can be checked to see if they are properly combined.
- **Semantic Analysis:** Once a sentence has been parsed using syntactic analysis, the semantics, or meaning, of the sentence is to be found. This is a very difficult task as it requires the knowledge of the world. The structures created by the syntactic analyzer are assigned meanings. A mapping is made between the syntactic structures and objects in the task domain.

Having gathered the preliminary knowledge of the specific domains relevant to this thesis, we can now move further to the more technically inclined literature. The next chapter takes a bird's eye view of the research work going on in the domain of the thesis.

Chapter 3

Literature Review

The purpose of the thesis presented here to bridge the gap between informal and formal specification of requirements. In this chapter we review different techniques proposed by researchers to convert system requirement specifications written in natural language to a formal notation, to move towards more rigorous way of requirement engineering and for natural language representation of formal methods for better user understanding.

3.1 Natural Language Representation of Formal Specification Models

Salek et al in [6] describes the architecture of the REVIEW system, which forms a part of the Metaview meta-system for capturing requirements information. The following figure shows the architecture of REVIEW system.

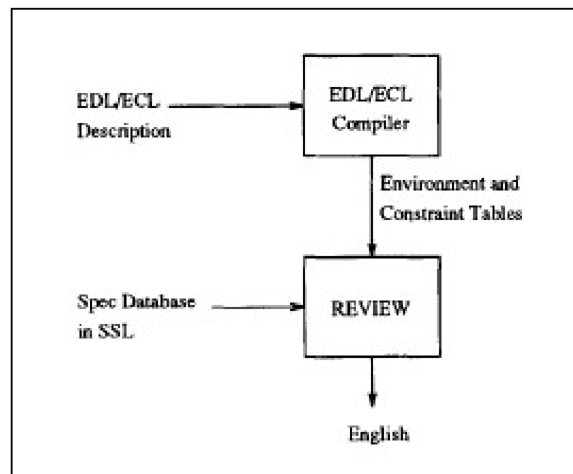


Figure 3.1: Architecture of REVIEW system

The EDL/ECL compiler produces environment tables which are used to instantiate the text generator for a particular environment. After this point, the REVIEW generator is detached from the rest of the Metaview system. It acts independently, as a tool specific to the environment the analyst has chosen. The SSL program is captured

in a specification database, which forms the input to REVIEW. The output is an English text description of the specification. A similar system for object oriented systems using OMT is described in [4].

Burke D.A. [14] describes a tool for linking formal specifications of programs, written in OCL, to informal specifications written in natural language, e.g. English or German. The core of this work is grammars written in the GF (Grammatical Framework) formalism. The long-term goal is to have a working tool integrated into the KeY system [18]. A library of GF operations for typical constructions of user-defined vocabulary is proposed. This allows domain specific vocabulary to be created for use in the natural language translations. Reasonable defaults are included when there is no user input. The input is a specification written in OCL. HTML and LaTeX formatting styles are introduced as part of the grammar and are used to structure the text in a suitable manner. Furthermore, the quality of translated text is improved by modifying the natural language linearization to produce more natural text. The translation process is shown in the following figure.

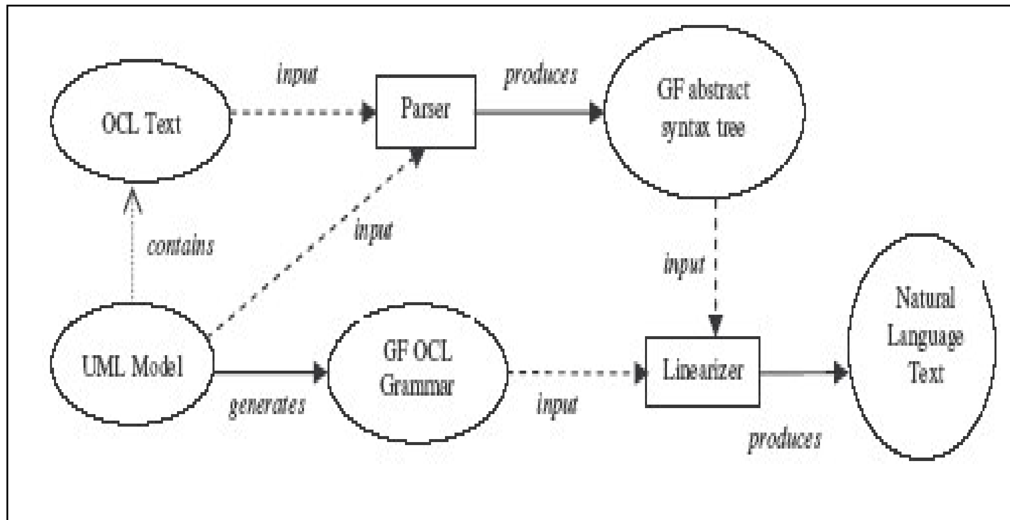


Figure 3.2: Translating OCL to Natural Language

In the next section we take a look on the attempts to convert a natural language specification to a formal representation.

3.2 Converting Natural Language Specification to Formal Notation

Fernandes R. and Cowie A.J. [24] propose a methodology for conversion of natural language and graphical representation of requirements to a model expressed in process algebra formalism. Natural language requirements are converted into process algebra using parsing and transformations. This parsing and transformation is reconfigurable in nature. Graphical requirements in the form of finite state diagrams are changed into textual form in natural language and then are converted to formal models using parsing and transformation. The formal methods can be analysed by the requirement engineer to make the system better. This results in iterative refinement of the natural language requirements and the tool configuration. The main aim the authors have behind this work is to reduce the rework required when formal methods are used early in the development life cycle.

The authors have developed a tool called as Borg. This system converts the requirements written in natural language to a formal syntax. First of all the requirements are converted to a T-model using the parsing. The parser used for parsing is Cico. The stored parse trees are mechanically transformed into refined intermediate models by a group of modellers. These models are then used by a projector to build an abstract model of the system. Finally, the abstract model is converted to a concrete formal method notation. Figure 3.3 shows the transformation method.

The formal model generated as a result of Borg can be analysed using an analysis tool called Nova. The implementation of the tool supports only the translation of transition based systems to the formal notation. These systems can be either in the form of natural language description or finite state machines. The finite state machine descriptions are converted to natural description before converting them to the process algebra formalism.

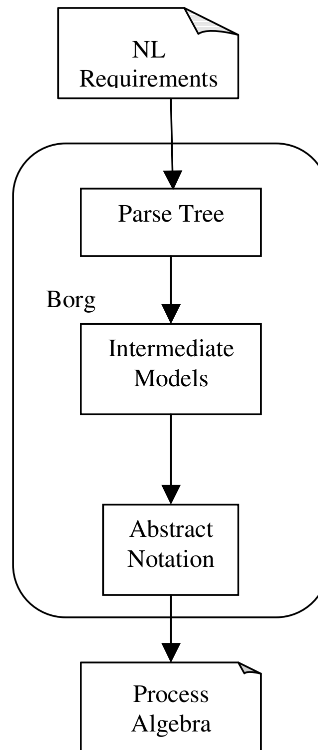


Figure 3.3: The Borg Tool Overview

A technique presented by Fraser et al [19] aims at bridging the gap between the formal and informal representation of software system requirements. The authors state that the two approaches for system specification are complementary rather than competing. The informal method of representation used is Structured Analysis, whereas the Vienna Development Method (VDM) is used as surrogate for the formal representation. Two methods are described for conversion of structured analysis model represented in the form of DFD to VDM. The first approach is conversion of structured analysis to VDM by a human analyst, with the use of structured analysis specifications as a cognitive guide to the understanding and structuring of the system. Specifications in VDM are developed by repeatedly refining complex specifications into sub-specifications until they are close to an implementable specification. In this paper, the refinement is based on the hierarchical partitioning of DFD. The authors assume that in structured analysis specifications, control processes are identified as while-structures, and the bottom level processes are described through decision tables. The decision tables are converted to VDM specifications using the decision table conversion rules. Then the specifications are composed using a bottom-up fashion

using the sequence composition rule. The payroll system is used as an example system showing conversion of structured specification to VDM.

Miriyala K and Harandi M. T. [60] see the derivation of formal specifications as a problem solving process. They describe a system for deriving formal specifications of data types and programs from the informal description. The system is an interactive system known as “SPECIFIER”. The problem solving techniques such as schemas, analogy and difference based reasoning are used in the system for derivation of formal specifications. The input or informal description is in restricted subset of natural language. It acts as an intelligent assistant to the requirement analyst and prompts for missing information and clarification of ambiguous concepts. If the informal description is a commonly occurring operation for which the system has a schema, then the formal specification is derived by instantiating the schema. If there is no such schema, the system tries to find a previously solved problem which is analogous to the current problem. If the problem thus found is in direct analogy with the current program, it applies and analogy mapping to obtain a formal specification. If the analogy found is only approximate, it solves the directly analogous part by analogy and then applies difference based reasoning to the remaining part. The two examples considered in the paper are of stack and sets. The architecture of the Specifier system is shown in the figure 3.4.

The process of specification derivation starts with an informal definition of the problem, expressed using restricted natural language. The pre-processor parses this informal specification and extracts important concepts occurring in it. The pre-processor then retrieves structure templates of the recognised concepts from the knowledge base. The schema based reasoner attempts to find schema for current problem, if it doesn’t find one, the analogy reasoner tries to find a past problem analogous to the current problem. The specifications are passed to post processor that simplifies the axioms in specifications and ensures that the final representation conforms with the desired syntax.

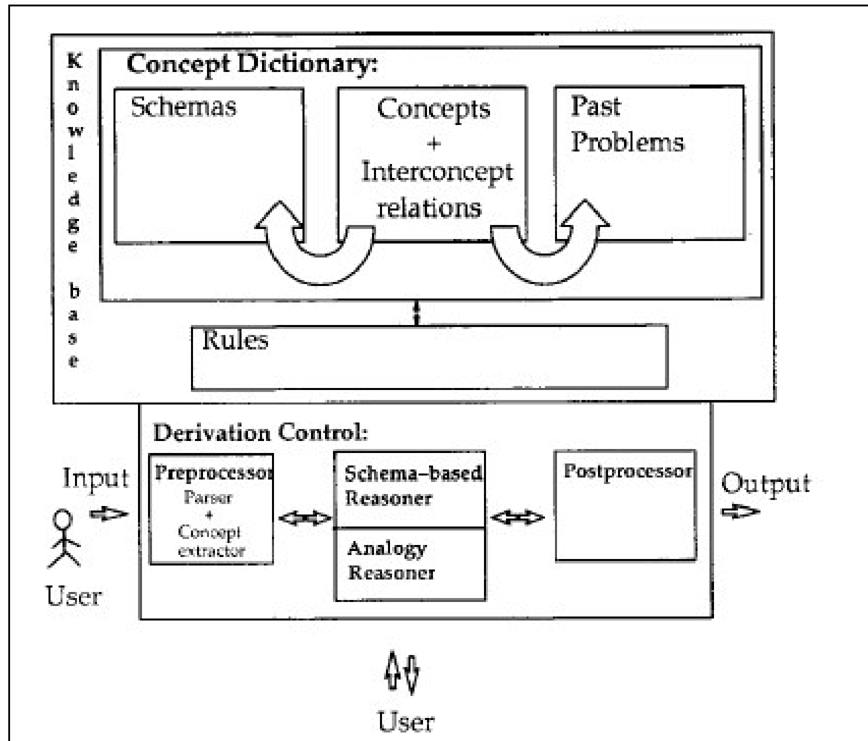


Figure 3.4: Schematic View of SPECIFIER system

Osborne M. and MacNish C. K. [1] propose a system using natural language processing to derive formal descriptions from descriptions expressed in controlled natural language. The authors have suggested extensions to some existing tools to provide a platform for detecting and resolving ambiguities. Doing some additions to controlled languages, the authors develop a system called Newspeak. The basis system taken is Alvey Natural Language Toolkit (ANLT), which is based upon the Longman's Dictionary of Contemporary English. The additions done in ANLT include the parse selection mechanism, which helps removing contextually inappropriate parses of the sentence and ranking these parses in terms of plausibility. The existing parser does not indicate when it cannot parse a sentence, so an error diagnostic system is introduced which gives reason for not being able to parse a given sentence. Also a resource binding is introduced to avoid the parser spending too much time on parsing a sentence. In implementation, first the mechanism was trained by parsing approximately 750 sentences that were supplied with the ANLT. For each sentence the preferred parse out of those parses produced was selected. Then it was recorded which rules were used in that parse, and the frequency of each rule's

application in the parses was noted. The frequencies were normalised against the total number of rule applications.

Now, during parsing, parse trees were scored by taking the product of the score of the rules used in each tree. For rules used in the tree that were not encountered in the training set of parses, a small value is assigned. To avoid underflow, the use of logarithms in place of the normalised scores was done, and then simply taking the sum of logarithms. For error diagnosis, the fragments of partially parsed sentence are combined together and repaired. To make the system resource bound, a limit is placed on the production of number of edges, i.e. the data structures used for parsing. If the number of edges exceeds that particular value, the system will halt. The authors also provide a case study of refining the requirements using the developed Newspeak system. The example system considered is the Landing Control System. A Goal-Structure framework is used as the basis of SRS, which is an acyclic graph of structured objects. Each object is either a goal (a decomposable property of the system that we would like to achieve), an effect (similar to a goal, but not necessarily desirable), a fact (a "bottom-level" property which will not be decomposed further), or a condition (similar to a fact, but only holding in particular scenarios). A natural representation for goals, effects, facts, conditions and their associated information is a frame, a data structure commonly used for representing hierarchical information in AI

The system helps in refining the SRS and helps in detecting ambiguities in the SRS.

Nelken R. and Francez N. [61] proposed a method for automatically translating natural language specifications into temporal logic. Using this method, users may express complex specifications in relatively free natural language, allowing multi-sentence specifications, the use of pronouns instead of repeating the description of previously mentioned objects and complex temporal relations. These specifications are translated into temporal logic, while ensuring the correctness of the translation. DRT, a linguistic theory of the semantic content of general natural language, is used as an intermediate representation level. The system represented in DRS (Discourse Representation Structure) is converted to a temporal logic called as ACTL. The system for translation of specification is named as SpecTran.

Alexander Holt [62] suggests guidelines for development, when natural language is used as the primary interface in the formulation of system specifications. An

experimental system which integrates a model checker for temporal logic with a natural language processing application is described. The core of this system is a converter that translates (a formal representation of the meaning of) specification sentences written in English to formulas of temporal logic, suitable for input to the model checker. A prototype system is built which integrates a natural language understanding component with the freely available SMV model checker program. The result is a tool which allows the formal verification of digital circuits using specifications expressed in English. SMV requires specifications to be written in the temporal logic CTL (computation tree logic). A parser for English, returning general-purpose semantic representations, is allied with a converter from these representations to CTL. The converter is connected to the SMV model checker, so that inferential information may be used during semantic interpretation.

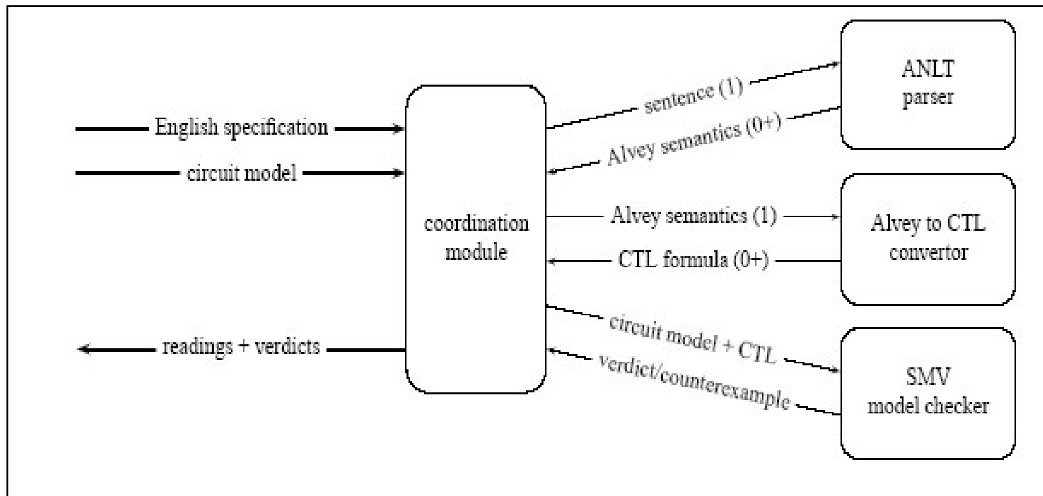


Figure 3.5: System Structure [62]

Cyre et al [63] describe two approaches to the automatic generation of behavioural VHDL models from descriptions written in natural language. Both approaches are based on a modelling style in which behaviour is represented by a system of interconnected processes. The first approach employs a semantic grammar to directly generate a single VHDL process from a paragraph written in a restricted English called ModelSpeak. The second approach accepts more general English and generates models consisting of multiple processes.

Lee B.S. and Bryant B.R. describe a system to convert natural language specification to formal representation in VDM++. In [10] they make use of

Contextual Natural Language Processing (CNLP) to handle the ambiguity problem in NL and Two Level Grammar (TLG) to deal with the different formalism level between Natural Language and formal language. In [9] they use DAML to deal with implicit domain knowledge to achieve automated conversion of natural language requirements to formal ones. First the requirements are represented in XML, then using CNLP, ambiguities are resolved. Further TLG is generated and finally the system is converted to VDM++. A similar approach is given in [23]. In all their approaches, the authors use a bank ATM example to elaborate the system and the conversion process. The following figure shows the conversion process as described in [8].

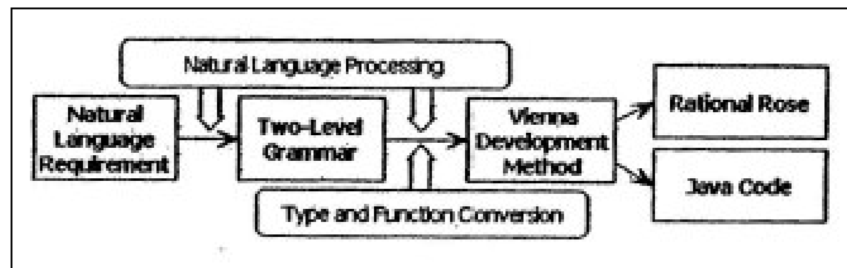


Figure 3.6: The conversion process [8]

In the next chapter, a system for conversion of natural language specifications to formal representation in Z is proposed.

Chapter 4

Proposed System and Implementation

A direct appeal to formal specification has not solved the problems arising due to the use of natural language for specifying the system. This is partly because of the restrictiveness and lack of habitability of formal languages [1]. An alternative approach, described in this thesis, is to use natural language processing (NLP) techniques to aid the development of formal descriptions from requirements expressed in natural language

Two of the most important characteristics of specifications are:

1. Specifications must be unambiguous and
2. Specifications must be understandable by the user. [55]

The unambiguous nature of software specifications can be obtained by expressing or representing the specifications using formal methods. But the requirements written in a formal language are not understandable by the customer. Thus if we want to keep this as a characteristic of our software requirements specification, we will have to represent them in the natural language, that the customer can understand easily.

4.1 Problem Statement

The thesis aims at bridging the gap between informal and formal specifications of software systems. The aim of proposed work is not to eliminate user's involvement, but as an assistance for the developer. The objectives of the thesis are:

- To show the feasibility of converting natural language specification of a system to the formal notation
- Facilitating the conversion of natural language specifications to formal specifications, thus bridging the gap
- Enabling the people who are not well versed with formal methods, to use formal methods to some extent and hence benefit from them, by

providing a conversion framework from natural language to formal specifications

- Development of a tool using NLP, to automate conversion of natural language specifications to formal ones
- Facilitating the use of natural language documents for requirements engineering purposes and communicating with non-technical users, and the use of formal methods for modelling, implementation and verification of the system.

4.2 Why is it worthwhile to address the problem

There are efforts going on to convert formal specifications to natural language, so that these are easily understood by customers, as described in section 3.1. There have been few attempts to automate the translation from a requirement document written in natural language to formal specification language [8, 10]. The conversion of natural language specifications to formal language, if done manually, is prone to errors due to the high level of expertise required and this becomes more difficult to manage in specifying larger projects. The automatic conversion will not only ease-off the task but also provide uniformity even in the formal representation of specifications. The thesis tries to demonstrate the possibility of certain automation from the informal to the formal specifications, by means of a system, available to assist an expert in specifications. For this a process of formalisation is proposed.

4.3 Overview of the System

The proposed system, FREND, takes as input a specification document written in English language. The system is modelled to translate the constraints in the requirement specification document to Z notation. POS tagging is used to identify the entities and their attributes. Brill's Tagger [65] is used to generate the POS tags for the words. The brief working of Brill's Tagger is given in section 2.4. A detailed list of tags assigned to various parts of speech by Brill's Tagger is provided in Appendix B. The architecture of the system is shown in figure 4.1. This system called FREND (pronounced as 'Friend') will assist the specifier, and ease off his task of translating natural language specifications to formal notation. FREND stands for Formal Requirements Extraction from Natural language Description.

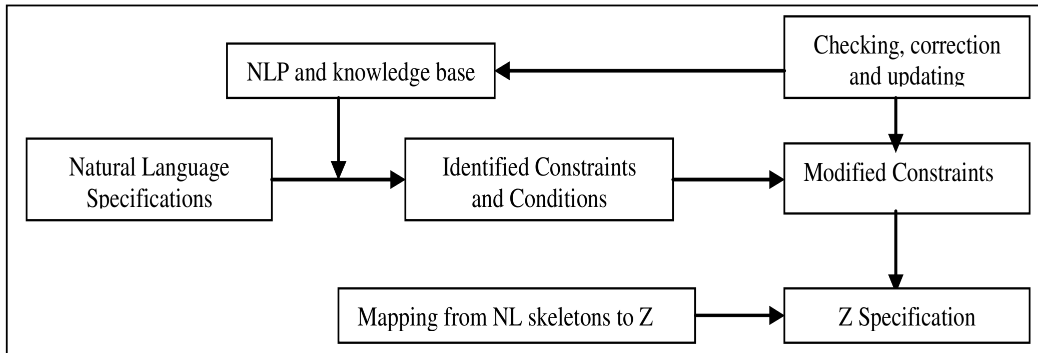


Figure 4.1: The architecture of the FREND system

4.4 Working of the System

The system works on natural language skeletons. Skeletons are particular sequences of various parts of speech. The constraints in the requirement specifications are captured by exploiting the use of certain words and phrases in the document and the grammatical properties of the natural language, which in this case considered is English.

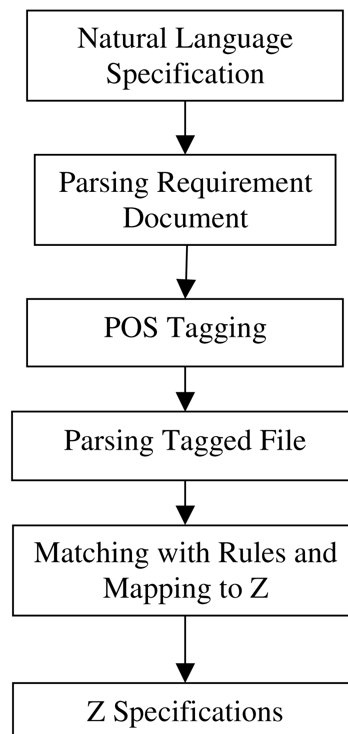


Figure 4.2: Working of FREND

The system has direct mapping from the inbuilt skeletons to Z notation. The evolutionary nature of the system allows new skeletons to be introduced or update existing ones, and their mapping to Z notation be specified. When a skeleton is

matched in the document, it is analyzed and converted appropriately to Z notation. For example, the skeleton

“<determiner> <noun> <helping verb> <verb> <adjective> <number> <noun>”

will fit both the following sentences:

1. “Manager can manage maximum two departments” and
2. “Student should take maximum three subjects”

or conversely, we can say that there will be same POS tagging generated for both the sentences.

4.5 Example System

The example shown here is to illustrate the conversion of natural language specifications to Z notation. The system under consideration is the Library Management System. The library system we have taken is the Central Library at TIET, still the requirements stated here are normally applicable to any public or institute library system. For the sake of simplicity, not all the aspects are modelled, rather only selected requirements are specified for the elucidation purpose. Also the lower level details are omitted from specifications.

Stated below are the requirements specified in natural language, which are then followed by the representation of these specifications in Z notation.

Natural Language Specifications:

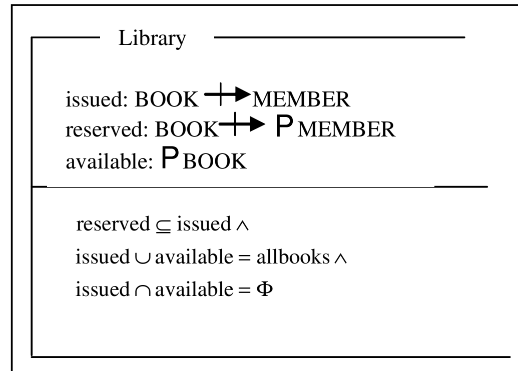
Consider a small library database with the following transactions:

1. *Issue a book to a member*
2. *Add a book to the library*
3. *Add a member to the library*
4. *Member reserves a book, if the book has been issued*

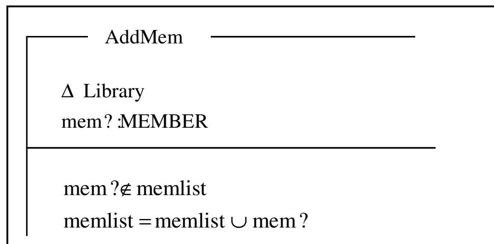
The database must also satisfy the following constraints

1. *All books in the library must be either available or issued*
2. *No book may be both issued and available*
3. *Only books that are issued can be reserved*

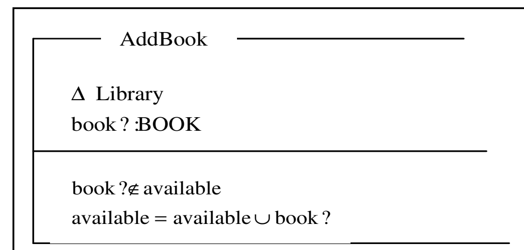
Z Specifications:



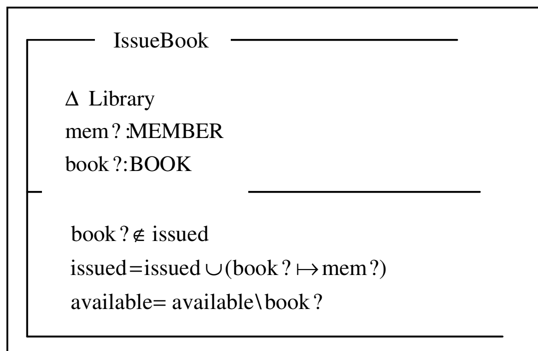
(A)



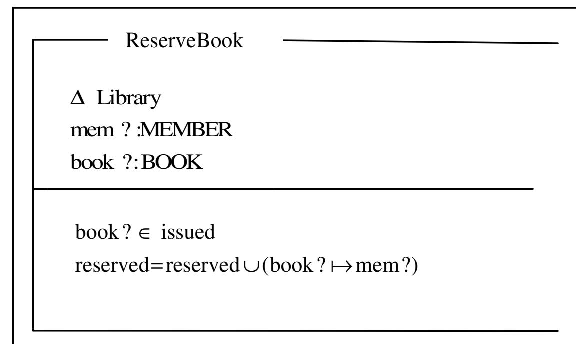
(B)



(C)



(D)



(E)

Figure 4.3: Example - Formal Specifications in Z

A subset of schemas produced for the system is shown above. The schemas are produced as follows.

1. The schema shown in figure (A) above is produced from all the constraints in natural language as follows:

    a. Constraint 1 is using double conjunction ‘either - or’, thus matching it with “either <adj> or <adj>”, a union of the two sets is shown to form that object, ‘all_books’

b. For constraint 2, the rule “No <noun> <verb> <prep> both <adj> and <adj>” is used and the intersection of two sets is declared as an empty set.

2. The addition rule is defined to take as input the object to be added. It first checks for object’s existence in the corresponding set and then updates the set by adding the object to the set. The result of this rule is shown in schemas (B) and (C) for adding member and book respectively.

3. “<Verb> <Noun> <Prep> <Noun>”, rule has a mapping from Noun1 to Noun2. This rule is used in schema (D) for issuing a book, as described in transaction 1.The addition rule is used to take input and update existing database.

4. For creating schema (E), we use rule “<noun> <verb> <noun>” to have mapping from Noun1 to Noun2. The second part of the sentence, “if the book is issued” is checked by testing membership of the input book in the set issued. Here addition rule is used for updating the set.

3.6 Features of System

This section describes some of the important features of the system. This can also serve as a user’s guide to the system, to some extent as screenshots of the implemented system are shown. The symbols to be used in Z notation are input using .jpg images.

- **Add rule:** To add a new schema rule, a user selects the sequence of POS tags, and then defines the corresponding Z notation for that. To input the parts of speech, combo boxes are provided. These combo boxes are loaded on runtime. To terminate the sentence and keep new combo box from loading, ‘.’ should be used. Figure 4.4 shows the partial screen for AddRule form. There are no restrictions placed on repetition of a specific sequence, i.e. more than one schema can be designed for same sentence or skeleton.

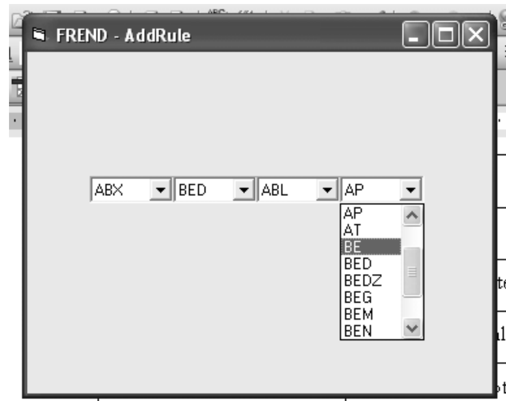


Figure 4.4: Partial View of AddRule Screen

- **Editing a Schema:** An already designed schema can be edited. The various editing options include – Renaming the schema, Addition / Deletion of invariants and declarations. An empty schema is provided to start with. A new schema can be developed using editing operations on this schema. Similarly, it can also be used to edit the generated schemas, during the review of formal notation. Figure 4.5 is a snapshot of the screen used for editing the schema.

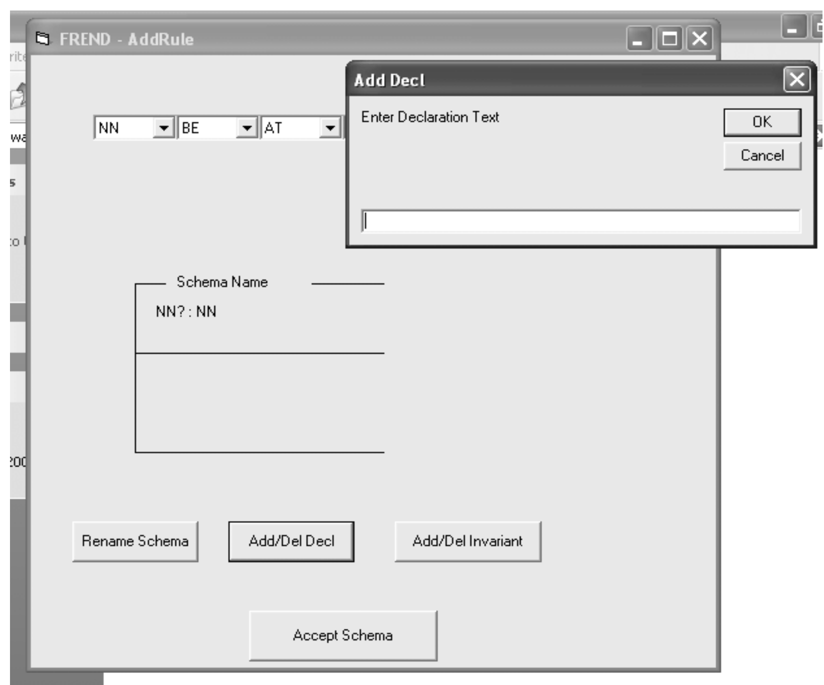


Figure 4.5: Editing/Adding a Schema

- **Conversion to Z schemas:** The system tags the input specifications file using Brill's Tagger. The tagger takes as input a file which is having one sentence per line. Thus the input file is parsed and converted to the desired format, by looking for sentence delimiters. The output of tagger is tagged file in a set of "word/tag" combinations. Thus the output file is again parsed to obtain only the tags part. The tag sequence is then matched with the rules defined. If more than one schema is designed for a sequence, the system displays all those schemas and then it is upto the user to choose an appropriate one. If there is no schema found, the system prompts it to user and demands for a schema to be specified. The output obtained can also be edited using edit schema options.

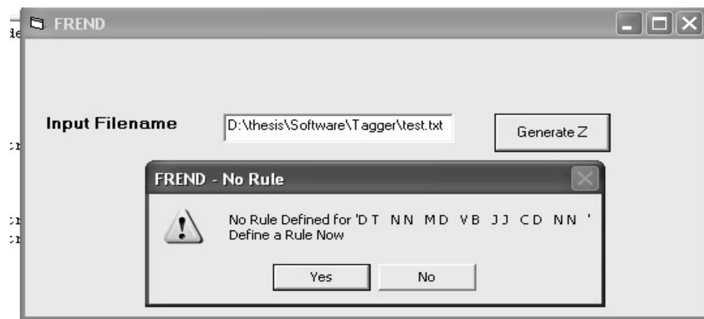


Figure 4.6: No schema found Screen

Chapter 5

Conclusions and Future Work

There is a wide gap between what formal methods can do and for what they are currently used. Though formal methods promise improved quality of software and partial automation of the software development process they are not readily accepted by domain specialists. Many domain specialists are not familiar or comfortable with formal notations and formal tool [53]. As a consequence, the great majority of software requirements continue to be specified in natural language, which is easy to use [54]. The thesis has tried to bridge the gap between formal and natural language specifications of a system. We present in this chapter the outcomes of the thesis and the future directions and scope.

5.1 Conclusions

In the thesis, a system for automated conversion of software requirement specifications written in natural languages to formal specifications in Z has been presented. The system is evolutionary in nature and enhances its knowledge base through an interactive system, using continual user input.

This system will help to a great extent to reduce the effort required for formal representation of specifications. The knowledge that is incorporated into the system by the experts can also be used by novices. Author does not claim the generated formal requirements to be 100% correct and review of the formal specifications produced is desired and recommended. Still human intervention is required to populate the knowledge base of the system. The purpose of the system is not to eliminate user's involvement, but as an assistance for the developer.

The system may be viewed as the one catering to lightweight formal methods. As stated in [64] the Lightweight formal methods often perform partial analysis on partial specifications only. They don't require much effort and cost to translate entire (informal) requirements documents into formal ones, nor to maintain formal and informal versions of specifications in parallel. This seems to be relevant in context of

the presented system. The system will have substantial benefits if incorporated in daily life of a requirement engineer.

5.2 Future Work

There is always a scope for improvement. The work in future will mainly need to address the weaknesses and shortcomings of the work done up to now.

- **Checking for the grammatically invalid constructs:** The system as now does not refrain the user from entering the constructs that are not grammatically valid in natural language. These constructs, however, never be encountered in real specification document, thus there is no need to define schemas for such constructs.
- **Integration with other tools:** The system can be integrated with other tools for better performance. E.g. a system for converting Z specifications to natural language can be used to convert to generate back natural language specifications for verification of correctness of conversion. Semantic analysers may also be required for the purpose. The formal output generated can be made to work with verification tools and with software reuse tools.
- **Output in a computer readable format:** The Z notation is easier to be read by human beings than computer. A more computer readable form can be generated.
- **Extension to other formal languages:** As a single formal language is not sufficient and appropriate for all (types of) specifications, conversion to other formal languages may be incorporated. Z may be used as an intermediate language, as translation of one formal language to other is easier than natural to formal conversion.
- **Making the system platform independent:** The current version of the system is designed to work with Windows operating system only. This may be extended for cross platform operations.

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]. A. Felfering , G. Fliedl and C. Kop , “Applying Natural Language Processing to Knowledge Based Configuration”, The 16th European Conference on Artificial Intelligence (ECAI-2004) August 23, 2004

[3]. I.J. Hayes and C.B. Jones, “Specifications are not (Necessarily) Executable”, Technical Report No. 148, Key Centre of Software Development, University of Queensland, Australia, January 1990.

[4]. J.M. Punshon, J.P. Tremblay, P.G. Sorenson and P.S. Findeisen, “From Formal Specifications to Natural Language: A Case Study”, Proceedings of 12th IEEE International Conference on Automated Software Engineering, 1-5 November 1997, pp:309 - 310

[5]. J.M. Wing, “A Specifier’s Introduction to Formal Methods”, IEEE Computer Volume 23, Issue 9, Sept. 1990, pp:8, 10 - 22, 24

[6]. A. Salek, P.G. Sorenson, J.P. Tremblay and J.M. Punshon, “The REVIEW System: From Formal Specifications to Natural Language”, Proceedings of the First International Conference on Requirements Engineering, 18-22 April 1994 pp. 220-229

[7]. K. Johannisson, “Formal and Informal Specifications”, PhD Thesis, Chalmers University of Technology, 2005

[8]. B.S. Lee and B.R. Bryant, “Automated Conversion of Natural Language Requirements to a Formal Specification Language”, Proceedings of the 39th Annual ACM Southeast Conference, Athens, Georgia, USA, March 2001, pp. 160-161

[9]. B.S. Lee and B.R. Bryant, “Contextual Natural Language Processing and DAML for Understanding Software Requirements Specifications”,

Proceedings of the 19th International Conference on Computational Linguistics, Taipei, Taiwan, August 2001, pp. 516-522

[10]. B.S. Lee and B.R. Bryant, “Contextual Knowledge Representation for Requirements Documents in Natural Language”, Proceedings of the 15th International FLAIRS Conference, Pensacola, Florida, USA, May 2002, pp. 370-374

[11]. E. Kamsties, D.M. Berry and B. Paech, “Detecting Ambiguities in Requirements Documents using Inspections”, Proceedings of the First Workshop on Inspection in Software Engineering (WISE'01), Paris, France, 23 July 2001, pp. 68 - 80.

[12]. B.S. Lee and B.R. Bryant, “Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language”, Proceedings of SAC 2002, Symposium on Applied Computing, Madrid, Spain, March 2002, pp. 932-936

[13]. B.R. Bryant and B.S. Lee, “Two-Level Grammar as an Object-Oriented Requirements Specification Language”, Proceedings of the 35th Hawaii International Conference on System Sciences Waikoloa, Hawaii, USA, January 2002

[14]. D.A. Burke, “Improving the Natural Language Translation of Formal Software Specifications”, Master’s Thesis, Complex Adaptive Systems International Masters Programme, Chalmers University of Technology, SE-412 96 Gothenburg, Sweden, December 2004

[15]. B.S. Lee and B.R. Bryant, “Prototyping of Requirements Documents Written in Natural Language”, Proceedings of the South-eastern Software Engineering Conference Huntsville, Alabama, USA, April 2002

[16]. B.S. Lee, “Automatic Transformation of Natural Language Requirements into Formal Specifications”, Proceedings of Doctoral Workshop of the 5th IEEE International Symposium on Requirements Engineering, Toronto, Ontario, Canada, August 2001

[17]. E.A. Strunk, “The Role of Natural Language in a Software Product”, MSc Thesis, University of Virginia, May 2002

[18]. R. Hahnle, K. Johannisson and A. Ranta, "An Authoring Tool for Informal and Formal Requirements Specifications", Fundamental Approaches to Software Engineering, volume 2306 of LNCS, Springer, 2002, pp: 233-248

[19]. M.D. Fraser, K. Kumar and V.K. Vaishnavi, "Informal and Formal Specification Languages: Bridging the Gap", IEEE Transactions on Software Engineering, Vol. 17, No. 5, May 1991, pp. 454-466

[20]. J.C. Knight, C. L. DeJong, M.S. Gibble and L.G. Nakano, "Why Are Formal Methods Not Used More Widely", The Fourth NASA Langley Formal Methods Workshop, September 1997, pp: 1-12,

[21]. A.D. Fuxman, "Formal Analysis of Early Requirements Specifications", MSc Thesis, University of Toronto, 2001

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

[23]. B.S. Lee and B.R. Bryant, "Automation of Software System Development Using Natural Language Processing and Two-Level Gramma", Proceedings of Monterey Workshop, Venice, Italy, October 2002

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

[25]. N. Castell and A. Hernandez, "Filtering Software Specifications Written in Natural Language", Proceedings of the 7th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence, 1995, pp: 447 - 455

[26]. "Software Requirements Engineering", SEI Interactive, March 1999 (http://www.sei.cmu.edu/interactive/Features/1999/March/Introduction/intro_mar99.htm)

[27]. K.E. Wiegers, "Writing Quality Requirements", Software Development, May 1999, pp: 44 - 48

[28]. A. Bucchiarone, S. Gnesi and P. Pierini, "Quality Analysis of NL Requirements: An Industrial Case Study", Proceedings of the 13th IEEE

International Conference on Requirements Engineering, 29 Aug.-2 Sept. 2005, pp: 390 - 394

[29]. M. Mannion and B. Keepence, “SMART Requirements”, ACM SIGSOFT Software Engineering Notes vol. 20 no. 2 April 1995, pp. 42-47

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

[31]. P. Rayson, R. Garside and P. Sawyer, “Assisting Requirements Engineering with Semantic Document Analysis”, Proceedings of RIAO 2000 (Recherche d'Informations Assistie par Ordinateur, Computer-Assisted Information Retrieval) International Conference, Collge de France, Paris, France, April 12-14, 2000, pp. 1363 - 1371.

[32]. 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

[33]. K.S. Hanks, J.C. Knight and Elisabeth A. Strunk, “Erroneous Requirements: A Linguistic Basis for Their Occurrence and an Approach to Their Reduction”, Proceedings of the 26th Annual NASA Goddard Software Engineering Workshop, December 2001, pp: 115

[34]. R.S. Pressman, “Software Engineering – A Practitioner’s Approach”, 5Ed, McGraw Hill, 2001, 0-07-118458-9

[35]. K. Li, R.G. Dewar and R.J. Pooley, “Computer-Assisted and Customer-Oriented Requirements Elicitation”, Proceedings of the 13th IEEE International Conference on Requirements Engineering, 2005, pp:479 - 480

[36]. 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

[37]. C.H. Wang, W.C. Chu, F.J. Wang, “Providing a Behavioral and Static Formal Model to Elicit the Functional Software Requirement”, IEEE International Conference on Information Reuse and Integration, 15-17 Aug. 2005, pp:44 - 49

[38]. J. Siddiqi and M. Chandrashekaran, "Requirements Engineering: The Emerging Wisdom", IEEE Software, Volume 13, Issue 2, March 1996, pp. 15-19

[39]. 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.

[40]. 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

[41]. 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

[42]. A. Hussein, "Software Requirements Specification (SRS) Checklist", Notes for 1-Oct-98, University of Calgary (http://elearning.tvm.tcs.co.in/Rwi/Rwi/SRS%20-20Check%20List.pdf)

[43]. D. D. Dankel II, M. S. Schmalz and K. S. Nielsen, "Understanding Natural Language Software Specifications", The 14th International Avignon Conference, June 1994, Paris, France, Vol. 3, pp. 25 - 34.

[44]. C. Grover, A. Holt, E. Klein and M. Moens, "Description of restricted natural language, ESPRIT LTR Project PROSPER (26241)", part of deliverable D5.1b University of Edinburgh, 1999

[45]. D.M. Berry and E. Kamsties, "The Syntactically Dangerous All and Plural in Specifications", IEEE Software, January-February 2005, pp. 55-57

[46]. 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

[47]. J.M. Spivey, "The Z Notation: A Reference Manual", 2Ed, Prentice Hall International, 1998

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

[49]. 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

[50]. J.M. Wing, "A Study of 12 Specifications of the Library Problem", IEEE Software, Volume 5, Issue 4, July 1988, pp:66 - 76

[51]. "Object Constraint Language Specification", version 1.1, Rational Software, 1 September 1997

[52]. G. Ammons, "STRAUSS: A Specification Miner", PhD Thesis, University of Wisconsin, Madison, 2003

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

[54]. N.E. Fuchs, U. Schwertel and S. Torge, "Controlled Natural Language Can Replace First-Order Logic", Proceedings of 14th IEEE International Conference on Automated Software Engineering, 1999, pp: 295-298

[55]. D. Patridge, "Artificial Intelligence and Software Engineering – Understanding the promise of Future", Glenlake Publishing Company Ltd, 1998

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

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

[58]. J.M. Spivey, "An Introduction to Z and Formal Specifications", Software Engineering Journal, January 1989, pp. 40-50

[59]. E. Rich and K. Knight, "Artificial Intelligence", 2Ed; Tata Mcgraw-Hill; 1991

[60]. K. Miriyala and M.T. Harandi, "Automatic Derivation of Formal Software Specifications from Informal Descriptions", IEEE Transactions on Software Engineering, Vol. 17, No. 10, October 1991, pp. 1126-1142

[61]. R. Nelken and N. Francez, “Automatic Translation of Natural Language System Specifications into Temporal Logic”, Proceedings of the 8th International Conference on Computer Aided Verification, Berlin, 1999, pp:360-371

[62]. A. Holt, “Formal Verification with Natural Language Specifications: Guidelines, Experiments and Lessons So Far”, South African Computer Journal, No 24, November 1999, pp. 253–257

[63]. W. R. Cyre, J. R. Armstrong, M. Manek-Honcharik, and A. Honcharik, “Generating VHDL Models from Natural Language Descriptions”, Proceedings of Euro-VHDL, Grenoble, France, September 19-23, 1994, pp:474-479

[64]. V. Gervasi1 and B. Nuseibeh, “Lightweight validation of natural language requirements”, Software – Practice and Experience, 2002, 32 pp. 113-133

[65]. Brill’s Tagger available at http://www.cs.jhu.edu/~brill/RBT1_14.tar.Z

Appendices

Appendix A: Glossary of Z Notation

Names

a,b	identifiers
d,e	declarations (e.g., $a : A;\ b, ... : B...$)
f,g	functions
m,n	numbers
p,q	predicates
s,t	sequences
x,y	expressions
A,B	sets
C,D	bags
Q,R	relations
S,T	schemas
X	schema text (e.g., d, $d \mid p$ or S)

Definitions

$a == x$	Abbreviated definition
$a ::= b \mid ...$	Data type definition (or $a ::= b\langle\langle x\rangle\rangle \mid ...$)
$[a]$	Introduction of a given set or basic type (or $[a, ...]$)
$a_$	Prefix operator
$_a$	Postfix operator
$_a_$	Infix operator

Logic

$true$	Logical true constant
$false$	Logical false constant
$\neg\, p$	Logical negation
$p \wedge q$	Logical conjunction
$p \vee q$	Logical disjunction
$p \Rightarrow q$	Logical implication ($\neg\, p \vee q$)
$p \Leftrightarrow q$	Logical equivalence ($p \Rightarrow q \wedge q \Rightarrow p$)

$\forall X \bullet q$	Universal quantification
$\exists X \bullet q$	Existential quantification
$\exists_1 X \bullet q$	Unique existential quantification
$\textbf{let}\ a == x;\ \ldots \bullet p$	Local definition

Sets and expressions

$x = y$	Equality of expressions
$x \neq y$	Inequality $(\neg\ (x = y))$
$x \in A$	Set membership
$x \notin A$	Non-membership $(\neg\ (x \in A))$
$\emptyset$	Empty set (also $\{\}$)
$A \subseteq B$	Set inclusion or subset
$A \subset B$	Strict set inclusion or subset $(A \subseteq B \wedge A \neq B)$
$\{x, y, \ldots\}$	Set of elements
$\{X \bullet x\}$	Set comprehension
$\lambda X \bullet x$	Lambda-expression – function
$\mu X \bullet x$	Mu-expression – unique value
$\textbf{let}\ a == x;\ \ldots \bullet y$	Local definition
$\textbf{if}\ p\ \textbf{then}\ x\ \textbf{else}\ y$	Conditional expression
$(x, y, \ldots)$	Ordered tuple
$A \times B \times \ldots$	Cartesian product
$\mathbb{P} A$	Power set (set of subsets)
$\mathbb{P}_1 A$	Non-empty power set
$\mathbb{F} A$	Set of finite subsets
$\mathbb{F}_1 A$	Non-empty set of finite subsets
$A \cap B$	Set intersection
$A \cup B$	Set union
$A \setminus B$	Set difference
$\bigcup A$	Generalized union of a set of sets
$\bigcap A$	Generalized intersection of a set of sets
$\mathit{first}\, x$	First element of an ordered pair
$\mathit{second}\, x$	Second element of an ordered pair
$\# A$	Size or cardinality of a finite set

Relations

$A \leftrightarrow B$	Relation ($\mathbb{P}(A \times B)$)
$a \mapsto b$	Maplet ((a, b))
$\mathrm{dom}\, R$	Domain of a relation
$\mathrm{ran}\, R$	Range of a relation
$\mathrm{id}\, A$	Identity relation
$Q \fatsemi R$	Forward relational composition
$Q \circ R$	Backward relational composition $(R \fatsemi Q)$
$A \lhd R$	Domain restriction
$A \ntriangleleft R$	Domain anti-restriction

$A \rhd R$	Range restriction
$A \ntriangleright R$	Range anti-restriction
$R(\| A \|)$	Relational image
$iter\, n R$	Relation composed n times
R^n	Same as $iter\, n R$
$R^\sim$	Inverse of relation
R^*	Reflexive-transitive closure
R^+	Irreflexive-transitive closure
$Q \oplus R$	Relational overriding ($(\mathrm{dom} R \ntriangleleft Q) \cup R$)
$a \underline{R}\, b$	Infix relation

Functions

$A \rightarrowtail\!\!\!\!\!\!+ B$	Partial functions
$A \rightarrow B$	Total functions
$A \rightarrowtail\!\!\!\!\!\!+ B$	Partial injections
$A \rightarrowtail B$	Total injections
$A \twoheadrightarrow\!\!\!\!\!\!+ B$	Partial surjections
$A \twoheadrightarrow B$	Total surjections
$A \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow B$	Bijective functions
$A \twoheadrightarrow\!\!\!\!\!\!\!+\!\!+ B$	Finite partial functions
$A \rightarrowtail\!\!\!\!\!\!\!+\!\!+ B$	Finite partial injections
$f\, x$	Function application (or $f(x)$)

Numbers

$\mathbb{Z}$	Set of integers
$\mathbb{N}$	Set of natural numbers $\{0, 1, 2, ...\}$
$\mathbb{N}_1$	Set of non-zero natural numbers $(\mathbb{N} \setminus \{0\})$
$m + n$	Addition
$m - n$	Subtraction
$m * n$	Multiplication
$m\ \mathsf{div}\ n$	Division
$m\ \mathsf{mod}\ n$	Modulo arithmetic
$m \leq n$	Less than or equal
$m < n$	Less than
$m \geq n$	Greater than or equal
$m > n$	Greater than
$succ\, n$	Successor function $\{0 \mapsto 1, 1 \mapsto 2, ...\}$
$m \ldots n$	Number range
$min A$	Minimum of a set of numbers
$max A$	Maximum of a set of numbers

Sequences

$\mathrm{seq}\, A$	Set of finite sequences
$\mathrm{seq}_1\, A$	Set of non-empty finite sequences
$\mathrm{iseq}\, A$	Set of finite injective sequences
$\langle\rangle$	Empty sequence
$\langle x, y, \ldots\rangle$	Sequence $\{1 \mapsto x, 2 \mapsto y, \ldots\}$
$s \frown t$	Sequence concatenation
$\frown/\, s$	Distributed sequence concatenation
$head\, s$	First element of sequence ($s(1)$)
$tail\, s$	All but the head element of a sequence
$last\, s$	Last element of sequence ($s(\#s)$)
$front\, s$	All but the last element of a sequence
$rev\, s$	Reverse a sequence
$squash\, f$	Compact a function to a sequence
$A \upharpoonleft s$	Sequence extraction ($squash(A \lhd s)$)
$s \upharpoonright A$	Sequence filtering ($squash(s \rhd A)$)
s prefix t	Sequence prefix relation ($s \frown v = t$)
s suffix t	Sequence suffix relation ($u \frown s = t$)
s in t	Sequence segment relation ($u \frown s \frown v = t$)
disjoint A	Disjointness of an indexed family of sets
A partition B	Partition an indexed family of sets

Bags

$\mathrm{bag}\, A$	Set of bags or multisets ($A \rightarrowtail\!\!\!\!\!+ \mathbb{N}_1$)
$[\![\,]\!]$	Empty bag
$[\![x, y, \ldots]\!]$	Bag $\{x \mapsto 1, y \mapsto 1, \ldots\}$
$count\, C\, x$	Multiplicity of an element in a bag
$C \sharp x$	Same as $count\, C\, x$
$n \otimes C$	Bag scaling of multiplicity
x in C	Bag membership
$C \sqsubseteq D$	Sub-bag relation
$C \uplus D$	Bag union
$C \cup\!\!\!\!- D$	Bag difference
$items\, s$	Bag of elements in a sequence

Schema notation

```
┌─S──────────
│ d
├──────────
│ p
└──────────
```

Vertical schema.

New lines denote ‘ ;’ and ‘$\wedge$’. The schema name and predicate part are optional. The schema may subsequently be referenced by name in the document.

d ——— p	**Axiomatic description.** The description may be non-unique. The predicate part is optional. The definitions apply globally in the document subsequently.
—— $[a, ...]$ —— d ——— p	**Generic construction.** The generic parameters are optional. The definitions must be unique. The definitions apply globally in the document subsequently.

$S \mathrel{\widehat=} [X]$	Horizontal schema
$[T; ... \mid ...]$	Schema inclusion
$z.a$	Component selection (given $z : S$)
θS	Tuple of components
$\neg S$	Schema negation
$S \wedge T$	Schema conjunction
$S \vee T$	Schema disjunction
$S \Rightarrow T$	Schema implication
$S \Leftrightarrow T$	Schema equivalence
$S \setminus (a, ...)$	Hiding of component(s)
$S \upharpoonright T$	Projection of components
$\mathrm{pre}\ S$	Schema precondition
$S \fatsemi T$	Schema composition (S then T)
$S \gg T$	Schema piping (S outputs to T inputs)
$S[a/b, ...]$	Schema component renaming (b becomes a, etc.)
$\forall X \bullet S$	Schema universal quantification
$\exists X \bullet S$	Schema existential quantification
$\exists_1 X \bullet S$	Schema unique existential quantification

Conventions

$a?$	Input to an operation
$a!$	Output from an operation
a	State component before an operation
a'	State component after an operation
S	State schema before an operation
S'	State schema after an operation
ΔS	Change of state (normally $S \wedge S'$)
ΞS	No change of state (normally $[S \wedge S' \mid \theta S = \theta S']$)
ΦS	Partial specification of an operation
$\vdash p$	Theorem
$d \vdash p$	Theorem with declarations ($\vdash \forall d \bullet p$)

Appendix B: POS Tags

Different tags assigned to parts of speech by the Brill's Tagger are shown below

Tag	Part of Speech
.	sentence closer (. ; ? *)
(	left parenthesis
)	right parenthesis
*	not, n't
--	dash
,	comma
:	colon
ABL	pre-qualifier (quite, rather)
ABN	pre-quantifier (half, all)
ABX	pre-quantifier (both)
AP	post-determiner (many, several, next)
AT	article (a, the, an)
BE	be
BED	were
BEDZ	was
BEG	being
BEM	am
BEN	been
BER	are, art
BEZ	is

Tag	Part of Speech
CC	coordinating conjunction (and, or)
CD	cardinal numeral (one, two, 2, etc.)
CS	subordinating conjunction (if, although)
DO	do
DOD	did
DOZ	does
DT	singular determiner/quantifier (this, that)
DTI	singular or plural determiner/quantifier (some, any)
DTS	plural determiner (these, those)
DTX	determiner/double conjunction (either)
EX	existential there
FW	foreign word (hypenated before regular tag)
HV	have
HVD	had (past tense)
HVG	having
HVN	had (past participle)
IN	Preposition
JJ	Adjective
JJR	comparative adjective
JJS	semantically superlative adjective (chief,top)
JJT	morphologically superlative adjective (biggest)
MD	modal auxiliary, helping verb (can, should, will)

Tag	Part of Speech
NC	cited word (hyphenated after regular tag)
NN	singular or mass noun
NN$	possessive singular noun
NNS	plural noun
NNS$	possessive plural noun
NP	proper noun or part of name phrase
NP$	possessive proper noun
NPS$	possessive plural proper noun
NR	adverbial noun (home, today, west)
OD	ordinal numeral (first, 2nd)
PN	nominal pronoun (everybody, nothing)
PN$	possessive nominal pronoun
PP$	possessive personal pronoun (my, our)
PP$$	second (nominal) possessive prounon (mine, ours)
PPL	singular reflexive/intensive personal pronoun (myself)
PPLS	plural reflexive/intensive personal pronoun (ourselves)
PPO	objective personal pronoun (me, him, it, them)
PPS	3rd. singular nominative pronoun (he, she, it, one)
PPSS	other nominative personal pronoun (I, we, they, you)
QL	qualifier (very, fairly)
QLP	post-qualifier (enough, indeed)
RB	adverb

Tag	Part of Speech
RBR	comparative adverb
RBT	superlative adverb
RN	nominal adverb (here, then, indoors)
RP	adverb/particle (about, off, up)
TO	infinitive marker to
UH	interjection, exclamation
VB	verb, base form
VBD	verb, past tense
VBG	verb, present participle/gerund
VBN	verb, past participle
VBZ	verb, 3rd singular present
WDT	wh- determiner (what, which)
WP$	possessive wh- pronoun (whose)
WPO	objective wh- pronoun (whom, which, that)
WPS	nominative wh- pronoun (who, which, that)
WQL	wh- qualifier (how)
WRB	wh- adverb (how, where, when)

Papers / Publications

1. Rocky Goyal and Rajesh K. Bhatia, “Automatic Conversion of Natural Language Software Specifications to Z Notation”, SERP ’06, International Conference on Software Engineering Research and Practice, Las Vegas, USA, June 26-29, 2006 **(Accepted)**