IMPROVEMENT IN THE REQUIREMENT SPECIFICATION USING Z LANGUAGE AND ITS SUCCESSOR

A thesis submitted in partial fulfillment of the requirements

for the award of degree of

Master of Engineering

in

Software Engineering

Submitted By

SATHISH KUMAR MIDDE

(Roll No. 800831007)

Under the supervision of

Mrs. SHIVANI GOEL

Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

THAPAR UNIVERSITY

PATIALA – 147004

June 2010

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Improvements in Requirement Specification Using Z language and Its Successor"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Shivani Goel and refers other researcher's works which are duly listed in the reference section.

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

M. Satish

(Sathish Kumar Midde)

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

Shivani Goel.

(Mrs. Shivani Goel)
Assistant Professor,
Computer Science and Engineering Department,
Thapar University, Patiala.

Countersigned by

12/07/10

(RAJESH BHATIA)
Professor & Head,
Computer Science & Engineering Department,
Thapar University,
Patiala.

R.K.Sharma 12/7/10

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

Acknowledgement

Words are simply not enough to express my gratitude towards to my guide **Mrs. Shivani Goel, Assistant Professor,** Computer Science & Engineering Department for her huge help to start thesis, guidance, making suggestions, encouragement all the time, and helped to finish this thesis successfully. She always provide a motivating and enthusiastic atmosphere to work with, it was a great pleasure to do this thesis under her supervision.

I am also thankful to **Dr. Rajesh Bhatia,** Head of Department, Computer Science & Engineering Department and **Mrs. Inderveer Channa,** P.G. Coordinator, for the motivation and inspiration that triggered me for the thesis.

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

Most importantly, I thank to my family, for giving me life in the first place, for educating me with aspects from both arts and sciences, for unconditional support and encouragement to pursue my interests, even when the interests went beyond boundaries of language.

Finally, my special thanks goes to authors whose works I have consulted and quoted in this work.

M. Sathish

Sathish Kumar Midde

(800831007)

Abstract

Z (pronounced Zed!) is a Formal Specification language that works at a high level of abstraction so that even complex behaviors can be described precisely and concisely, which is also used to specify the functional requirement of the system. A functional specification is a formal document used to describe in details for software developers a product's intended capabilities, appearance, and interactions with users.

The Z notation is a strongly typed, mathematical, specification language. It is not an executable notation; it cannot be interpreted or compiled into a running program. The Z notation is a model-based specification language based on set theory and first-order predicate logic. It provides rich data structures and facilities to define operations on them.

An abstract Z specification of an Automated Teller Machine (ATM) environment is defined in this thesis. This specification is used to represent the state of the world and the operations defined on it. The Z schemas are written by using Z/Word tool, supports all types of notations which are used in Z language. Then, attempt to ensure that some constraints on this system are not violated. For this, converted the Z specification into an Alloy model that can be put into the Alloy Analyzer for fully automatic analysis. Alloy is a formal language deeply rooted in Z. Alloy's writing mechanisms are designed to have the flexibility of Z's schema calculus, but are based on different idioms.

Combining Z specification to C++ code to provide a prototyping facility by integrating the execution of code into the interpretation of a specification. That is, it is not aimed as a code generation system, but kind of a tool for analyzing specification (including syntax, semantic and type checking) and for helping a software developer in obtaining code from specification.

Table of Contents

Certificate i
Acknowledgement ii
Abstract iii
Table of Contents iv
List of Figures vii
List of Tables viii
Chapter 1: Introduction 1
1.1 Formal Methods 1
1.1.1 Formal specification languages 2
1.2 Thesis Roadmap 3
Chapter 2: Literature Review and Related Work 4
2.1 Literature Review 4
2.2 Brief Introduction to Z 5
2.2.1 The Mathematical Language 5
2.2.1.1 Propositions of Algebra logic 5
2.2.1.2 Predicate Logic 6
2.2.1.3 Sets and Relations 8
2.2.1.4 Functions 11
2.2.1.5 Sequences 12
2.2.1.6 Bags 13
2.2.2 The Schema Language 13
2.2.2.1 Basic type Definitions 14
2.2.2.2 Axiomatic descriptions 14
2.2.2.3 Constraints 15
2.2.2.4 Schema Definition 15

2.3 Difficulties with Z……16

Chapter 3: Problem Statement 19

3.1 Introduction.....19

3.2 Objectives of Thesis.....20

Chapter 4: Case study 21

4.1 Introduction 21

4.1.1 Conceptual Model of the ATM 22

4.1.2 Requirement Analysis.....23

4.2 Z-specification (Formal Model of ATM) 23

4.2.1 Type Declarations 24

4.2.2 State Specification……...…… ……..……..25

4.2.3 Error Handling…… …..……..……...29

4.2.4 Event Handling……..……..32

4.3 Verification…….…….....36

Chapter 5: Alloy Model……...38

5.1 Motivation for Using Alloy …….……...38

5.1.1 Alloy's Main Features……..……...38

5.2 Basic Notations ……...39

5.2.1 Logic….……39

5.2.2 Language..…….……..…….……...41

5.2.2.1 Signature…….……..…….…….……41

5.2.2.2 Facts……..…….…..…..……42

5.2.2.3 Functions...……. …….……..…….……43

5.2.2.4 Module Header……..….…….…….……43

5.2.2.5 Predicates……..…..……..……..….……44

5.2.2.6 Assertion……..…..…….…….…...45

5.2.3 The Analysis: Check And Run Commands..... 45

5.2.3.1 Check command…...……..…….. ……...45

5.2.3.2 Run command……...……..……...46

5.3 Converting the Z specification into Alloy……..……...……46

Chapter 6: Combining Z specification to C++ code……..……55

6.1 Basic Class…….…….....56

6.2 C ++…….. ...57

6.3 Structural Mapping……57

6.3.1 Basic rules…….…… 57

6.3.2 Type Translation……..... 58

6.4 Architecture of Z-C++…...…… ……..…….……...58

6.4.1 An example of mapping…….....59

6.5 Implementation of a Mapping System…… ……..... 61

6.5.1 LaTeX mark-up …....…….…… 61

6.5.2 Litigate and scanning.....62

6.5.3 Lexing and Parsing..... 63

6.5.4 Mapping..... 65

Chapter 7: Conclusion and Future Scope……..... 66

7.1 Conclusion……...…… 66

7.2 Future Scope……...….....66

References…….…….....67

List of Publications.....70

List of Figures

Figures:

Figure 2.1: Structure of a Z Specification……..14
Figure 2.2 : Graph of time duration formula…….17
Figure 4.1: Conceptual model of ATM……...……..23
Figure 4.2: Type checking the Z specifications…….36
Figure 4.3: Validation of Z specification…… ……..36
Figure 5.1: Visualization view of NO card should be both credit and debit…… ….48
Figure 5.2: Next solution of NO card should be both credit and debit…….48
Figure 5.3: XML view of NO card should be both credit and debit…….49
Figure 5.4: Tree view of NO card should be both credit and debit……...49
Figure 5.5: Visualization view for invalid thumb extension…….……50
Figure 5.6: Uniqueness representation for Card-Thumb Mapping.....……..51
Figure 5.7: One menu option did not match with other……52
Figure 5.8: Map relation between card and thumb in data base1..……54
Figure 5.9: Map relation between card and thumb in data base2…….54
Figure 6.1: Basic idea of combining code and specification in the Toolbox…….…55
Figure 6.2: Z module specification…….…….....57
Figure 6.3: The architecture of the structural mapping of Z-C++…….59
Figure 6.4: Bank module…… .60
Figure 6.5: Mapping……..62
Figure 6.6: structure of name table.....63
Figure 6.7: Structure of Class Attribute……..…… …….……63
Figure 6.8:Translation of a specification……..…….…….…….....64
Figure 6.9: Overview of parser process..... 64

List of Tables

Table 2.1: Logical propositional connectives……..…..…6

Table 2.2: The truth table for the propositions…….…....6

Table 2.3: List of operators on Sets. …….…..….9

Table 2.4: Relation operators in Z language……..10

Table 2.5: Functions in Z language…….....11

Table 2.6: Brief description about domain and range of functions……12

Table 2.7: List of sequence notations in Z…….12

Table 2.8: List of Bags operators in Z……...13

Table 5.1: The logic set operators in Alloy. …….40

Table 5.2: List of Boolean operators…….40

Table 5.3: Unary operators……41

Table 5.4: Converting the Z specification into Alloy……46

Chapter 1
Introduction

Computer-based systems are increasingly integrated into day-to-day life. They either control or provide platforms for communication networks, transportation facilities, economic markets, health-care systems, and safety and security facilities. One of the most challenging tasks in software development is to assure reliability of systems being designed and constructed. This becomes even more important as the use of software increases dramatically in embedded systems within life-critical environments such as medicine, air traffic control and other transportation systems, spacecraft control, and national defence weapons deployment and activation. With the increasing complexity of these systems, effective design and development of high quality systems that conform to their requirements are extremely challenging and the costs of design faults and system failures are high. Proper understanding of the requirements, exact design decisions, and effectively communicating such decisions with the domain experts as early as possible will play an important roles in the design of complex systems. These challenges call for acceptance of proper engineering methods and tools and have motivated the use of formal methods in software engineering.

Formal methods provide a framework within which people can specify, develop and verify systems in a systematic rather than ad hoc manner. They give us techniques to formally specify the functionality of a system, to verify its correctness or to develop the system gradually from an abstract specification to its implementation. All these aspects are important when designing safety-critical systems.

1.1 Formal Methods

Formal methods refer to mathematically based techniques for the specification, development and verification of software and hardware systems [30]. Formal methods may be applied at a number of levels:

Formal Specification: A mathematical description of the desired behaviour of a software system specifying in abstract terms what a system should do and not how to do it. During the formal specification phase, the engineer strictly defines a system using a modelling language. Modelling languages are fixed grammars which allow

users to model complex structures out of predefined types. This process of formal specification is similar to the process of converting a word problem into algebraic notation.

Formal development and verification: A formal development process involves iteratively refining a formal specification to produce the finished system. Formal Methods differ from other specification systems by their heavy emphasis on provability and correctness. By building a system using a formal specification, the designer is actually developing a set of theorems about his system. By proving these theorems correct, the formal methods ensures the correctness of the system. The process of proving or disproving properties of the software system against a formal specification is known as formal verification.

Implementation: Once the model has been specified and verified, it is implemented by converting the specification into code. As the difference between software and hardware design grows narrower, formal methods for developing embedded systems have been developed.

1.1.1 Formal specification languages

Formal specification languages almost invariably use mathematically based syntax. This allows the user to create a high-level model of the system using mathematical structures such as sets, propositions and mappings. The mathematical notation provides a well-defined and well-known semantics, which help to prevent the vagueness and ambiguities which may arise in a more informal requirements gathering phase. Another advantage of using mathematical notation is the ability to apply logical inference rules for automatically validating the internal consistency of a model.

The expressiveness of mathematical notation allows a high level of abstraction allowing the user to focus on the core functionality of the system without prematurely worrying about implementation details.

A considerable number of formal specification languages are available, some of them are:

- **Z-** Pronounced "zed'. Based on Zermelo-Frankel set theory, Lambda Calculus and First Order Predicate Logic. Developed in 1977 by Jean-Raymond Abrail, Steve Schuman and Betrand Meyer it was later further developed at the programming research group at Oxford University.
- **B-** The B Method is based on Abstract Machine Notation. Developed by Jean-Raymond Abrail it allows easier refinement to code (compared to Z). A tool set is available supporting specification, design, proof and code generation [31].
- **RAISE – R**igorous **A**pproach to **I**ndustrial **S**oftware **E**ngineering. Developed by Dines Bjorner as part of the European ESPRIT II LaCoS project in the 1990s. The methods consist of a set of tools based on RSL, the Raise Specification Language [34].
- **Alloy-** Developed in 1997 by the Software Design Group at M.I.T. Alloy is a declarative formal specification language designed specifically for automatic analysis. The toolkit includes a model finder/ checker which convert the model to a Boolean formula which can then be checked using a SAT solver [35].

1.2 Thesis Roadmap

Chapter 1 gives introduction to Formal Methods and Formal Specification Languages. Chapter 2 gives the brief description about Z language, the various types of operators available in Z and what are the difficulties with it. Definition of duration calculus concepts as they apply to specification in Z and how it can be represented along with the schemas is given. Chapter 3 gives the introduction to the problems statement and the objectives of this thesis. In Chapter 4, An Automated Teller Machine (ATM) is a safety-critical and real time system is taken as a case study. The conceptual model of the ATM and corresponding Formal Model of ATM are defined. Special attention to find errors and solutions for those errors are done Chapter 5. In Chapter 5, introduction to the Alloy model, how Alloy is related to the Z, Alloy main features and solution for the errors are defined. Chapter 6 provides how the Z specification can be combined to Code. This basic idea will be helpful for the code generation process. Chapter 7 concludes the work done in this thesis.

Chapter 2
Literature Review and Related Work

2.1 Literature Review

The Z language is a state based, formal specification language that is based on Zermelo Fränkel axiomatic set theory and first order predicate logic. The Z notation [9, 10] is a strongly typed, mathematical, specification language. It is not an executable notation; it cannot be interpreted or compiled into a running program. There are some tools for checking Z texts for syntax and type errors in much the same way that a compiler checks code in an executable programming language. Z is a model-based notation. One of the other main ingredients in Z is a way of decomposing a specification into small pieces called schemas. The schema is the feature that most distinguishes Z from other formal notations. In Z, schemas are used to describe both static and dynamic aspects of a system. In [15], proposal to a structured-expandable format of a use case is given, which is expressed in Z notation and then represented visually using an Entity-Relationship diagram, which has become a literal industry standard for capturing requirements with Z notation and again represent it visually using ER diagram. Comparing the different specification methods theoretically and address their differences by designing a particular part of the ABM system using each method [7], the authors has confirmed that it is important to combine prototyping technique and formal methods for development of large scale systems. The Z specification describes the data model, system state and operations of the system. Importantly, it can serve as a single, reliable reference point for those who investigate the needs, those who implement programs to satisfy those needs, those who test the results, and those who write instruction manuals for the system [10].

A fuzzy logic tool kit has been developed for the formal specification language Z. It permits the incorporation of fuzzy concepts into the language while retaining the precision of any Z specification [17]. A framework for the specification of parallel and distributed real-time systems is presented by using the formal specification language Z [18]. This paper proposes a practical and systematic approach for the specification of real-time systems in Z. Many critical control systems are developed using formal methods. A method to safety-critical railway signalling systems for the

formal requirement specification using Z and analyzed the specification results [19]. The key components of Network Security Policy Model are formalized in order to be sharp, precise and prevent their multiple interpretations, represented the model using the well known formal notation, Z [20] . Z also helps in refinement towards an implementation by mathematically relating the abstract and concrete states. In the journal article "Formal Specification and Documentation of Microprocessor Instruction Sets" [22], author discusses the use of Z to define a microprocessor based system. Z is being used by a wide variety of companies for many different applications. Many institutions use Z because they choose to, rather than because it is mandated by a defence or security related client [22].

2.2 Brief Introduction to Z

In the Z notation there are two languages: the Mathematical Language and the Schema Language. The mathematical language is used to describe various aspects of a design: objects, and the relationships between them. The schema language is used to structure and compose descriptions: collating pieces of information, encapsulating them, and naming them for reuse.

2.2.1. The Mathematical Language

2.2.1.1 Propositions of Algebra logic

A logical language based upon traditional propositional calculus which is part of the logical language of Z. Each component of the language is presented alongside rules that explain when it may be introduced or eliminated. Collected together, these rules form a system of natural deduction: they state what may be derived from a proposition, and under what conditions that proposition may be concluded.

i) Propositional logic: Propositional logic deals with the statement of said facts which must be either true or false, but not both. Statements or verb assertions will denote by the letters p, q, r, s....etc (with or without subscripts). Some statements are composite, that is composed of sub statements and various connectives.

Example1: "Roses are red and violets are blue" is a composite statement with sub statements "Roses are red" and "violets are blue".

Example2: "where are you going?" is not a statement since it is neither true nor false.

In logical language, propositions may be connected in various ways. The following table describes five propositional connectives, arranged in descending order of operator precedence:

Table 2.1: Logical propositional connectives.

Operator	Meaning	Value
¬	Negation	Not
∧	Conjunction	And
∨	Disjunction	Or
⇒	Conditional	Implies
⇔	Biconditional	If and only if

Table 2.2: The truth table for the propositions.

P	Q	P∧Q	P∨Q	P⇒Q	P⇔Q	¬p
T	T	T	T	T	T	F
T	F	F	T	F	F	F
F	T	F	T	T	F	T
F	F	F	F	T	T	T

2.2.1.2 Predicate Logic

The propositional logic is not powerful enough to represent all types of assertions that are used in computer science and mathematics, or to express certain types of relationship between propositions such as equivalence.

For example, the assertion "x is greater than 1", where x is a variable, is not a proposition because we cannot tell whether it is true or false unless you know the value of x. Thus the propositional logic cannot deal with such sentences. Also the

pattern involved in the following logical equivalences cannot be captured by the propositional logic:
"Not all birds fly" is equivalent to "Some birds don't fly".
"Not all integers are even" is equivalent to "Some integers are not even".

A **predicate** is a verb phrase template that describes a property of objects, or a relationship among objects represented by the variables. There are two logical constants in predicate logic, namely true and false.

Example1: "The sky is blue", and "The cover of this book is blue" come from the template "is blue" by placing an appropriate noun/noun phrase in front of it. The phrase **"is blue"** is a predicate and it describes the property of being blue.

i) Quantification: Full predicate logic augments propositional logic with quantification over a list of variables X, including type information in the case of Z [23]:

- Universal quantification: $\forall$ X $\bullet$ q is only true when the predicate q is true for all possible values of X.
- Existential quantification $\exists$ X $\bullet$ q is true if there is any (at least one) set of values for X possible which make q true. There may be more than one value; indeed all possible values, as required for universal quantification, would still be valid.
- Unique existential quantification $\exists_1$ X $\bullet$ q is a special but useful case of the more general existential quantification which only allows X to take a single value, not an arbitrary (non-zero) number of values.

The scope of the variables listed in X is bounded by the clause q in the examples above. Thus the same names may be reused outside the clause to stand for different variables if desired.

ii) Laws: There is a rich set of algebraic laws for transforming predicates when performing proofs about specifications. A good selection of these is presented in [24].

2.2.1.3 Sets and Relations

In the Z notation, there are several ways of defining an object. Simply declare, may be define by abbreviation, or may define by axiom. In addition, there are special mechanisms for free types and schemas, discussed later.

i) Declarations: The simplest way to define an object is to declare it. If the object is a given set, or basic type, then we does this by writing its name between brackets: for example, the declaration

⟦Type⟧.

Introduces a new basic type called Type. 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.

The question arises, why fuss about types? They introduce a lot of extra complexity to a specification. However this proves to be well worthwhile for the following reasons:

1. It helps to structure specifications. Specify the set of possible values from which a variable can be drawn, and then further constrain the variable using a predicate if required. For example in the (English) statement: X is a path of least cost from A to B could be given X a type: *X: Path (A↦ B).*
2. It helps avoid meaninglessness specifications.

The use of types means the specification can more easily be checked for consistency, either manually by human inspection, or automatically using the machine assistance of a type-checker.

ii) Set: A fundamental concept of in all branches of mathematics is that of a set. A set is any well defined list, collection or class of objects. The objects in sets can be any thing: numbers, peoples, letters, rivers, etc. These objects are called the elements or members of the set.

ℙ-Power set: which is the set of all subsets of a set. The set of all finite subsets is a subset of the power set.

Table 2.3: List of operators on Sets.

Notation	Name	Definition
$\{I : X \mid p\}$	set comprehension	elements i of X such that they should satisfy p, by axiom of Separation
$\{i_1: X_1;..; i_n: Xn \bullet X\}$	set comprehension	$\{w : W \mid \exists\, i_1 : X_1 ; ...; i_n : X_n \bullet w = X\}$
$X_1 \cup X_2$	union	$\{i : W \mid i \in X_1 \vee i \in X_2\}$
$X_1 \cap X_2$	intersection	$\{i : X_1 \mid i \in X_2\}$
$X_1 \setminus X_2$	difference	$\{i : X_1 \mid i \notin X_2\}$
$\cup A$	Generalized union	$\forall A:\mathbb{P}(\mathbb{P}\ X) \bullet \cup A=\{x: X \mid (\exists S:A \bullet x \in S)\}$
$\cap A$	Generalized intersection	$\forall A :\mathbb{P}(\mathbb{P}\ X) \bullet \cap A=\{x:X \mid (\forall S:A \bullet x \in S)\}$
$X_1 \subseteq X_2$	Subset	$\forall\ X_1,X_2:\mathbb{P}X \bullet (X_1 \subseteq X_2 \Leftrightarrow (\forall x:X \bullet x \in X_1 \Rightarrow x \in X_2))$
$X_1 \subset X_2$	Proper subset	$\forall\ X_1, X_2 : \mathbb{P}X \bullet (X_1 \subset X_2 \Leftrightarrow (X_1 \subseteq X_2 \wedge X_1 \neq X_2)$
$X_1 = X_2$	Equality	$\forall X_1, X_2 \subseteq X\ ;\ X_1 = X_2 \Leftrightarrow (\forall\ x : X_1 \bullet x \in X_2\ ) \wedge\ (\forall\ y : X_2 \bullet y \in X_1\ )$
$X_1 \neq X_2$	Inequality	$\forall x,y : X \bullet x \neq y \Leftrightarrow \neg\ (x = y)$
$\mathbb{P}_1$	Non-empty subsets	$\mathbb{P}_1X = = \{\ S : \mathbb{P}X \mid S \neq \varnothing\ \}$
$\varnothing, \{\}$	empty set	$\{i : X \mid \text{false}\ \}$
$\{e\}$	singleton set	$\{i : X \mid i = e\ \}$

Sets will usually be denoted by capital letters A, B, C ...X, Y, Z. Define a particular set by actually listing its members, for example, let A consist of members 1, 3, 5, 7.

Then A = {1, 3, 5, 7}. That is, the elements are separated by commas and enclosed in brackets {}. This is called *tabular form* of a set.

Defining a particular set by stating properties which its elements must satisfy, for example, let B is the set of all even numbers, then use usually x, to represent an arbitrary element and write B={x | x is even or (x/2==0)} Which reads as "B is the set of numbers x such that x is even". This is called the *builder's form* of a set.

$\{\varnothing\} \neq \varnothing$. It is important to understand that the set containing the empty set is not the same as the empty set itself. It is a set containing one element (namely, the empty set).

iii) Relations: A relation is defined to be a set of Cartesian pairs. There are several operations involving relations, which are given equivalences in Table 2.4. If X and Y are sets, then $X \leftrightarrow Y$ denotes the set of all relations between X and Y. The relation symbol may be defined by generic abbreviation: $X \leftrightarrow Y = = P(X \times Y)$. Any element of $X \leftrightarrow Y$ is a set of ordered pairs in which the first element is drawn from X, and the second from Y: that is, a subset of the Cartesian product set $X \times Y$. The simplest examples are the domain and range functions, 'dom' and 'ran'. If R is a relation of type $X \leftrightarrow Y$, then the domain of R is the set of elements in X related to something in Y. The range of R is the set of elements of Y to which some element of X is related.

In a relation R, first R: first $(e_1, e_2) = e_1$; second R: second $(e_1, e_2) = e_2$.

Table 2.4: Relation operators in Z language.

Notation	Name	Definition
id f	identity function	λ i : f ● i
dom f	domain	{i : f ● first i}
$f^{\sim}$	relational inversion	{i : f ● second i}
$f_1 \lhd f_2$	domain restriction	$\{i : f_2 \mid \text{first } i \in f_1\}$
$f_1 \rhd f_2$	range restriction	$\{i : f_1 \mid \text{second } i \in f_2\}$
$f_1 \lhd\!\!\!- f_2$	domain subtraction	$\{i : f_2 \mid \text{first } i \notin f_1\}$
$f_1 (\!\mid f_2 \mid\!)$	relational image	$\{i : f_1 \mid \text{first } i \in f_2 \bullet \text{second } i\}$
$f_1 \oplus f_2$	relational overriding	$((\text{dom } f_2) \lhd\!\!\!- f_1 \cup f_2)$

2.2.1.4 Functions

Table 2.5: Functions in Z language.

Notation	Name	Definition
$X \pfun Y$	Partial function	$\{ f : X \rightarrow Y \mid (\forall x : X ; y_1, y_2 : Y \bullet (x \mapsto y_1) \in f \land (x \mapsto y_2) \in f \Rightarrow (y_1 = y_2) \}$
$X \pinj Y$	Partial injections	$\{ f : X \pfun Y \mid (\forall x_1, x_2 : \text{dom} f \bullet f(x_1) = f(x_2) \Rightarrow x_1 = x_2) \}$
$X \psurj Y$	Partial surjections	$\{ f : X \pfun Y \mid \text{ran } f = Y \}$
$X \fun Y$	Total functions	$\{ f : X \pfun Y \mid \text{dom } f = X \}$
$X \inj Y$	Total injections	$(X \pinj Y) \cap (X \fun Y)$
$X \surj Y$	Total surjection	$(X \psurj Y) \cap (X \fun Y)$
$X \bij Y$	Bijections	$(X \surj Y) \cap (X \inj Y)$

Description: If X and Y are sets, $X \pfun Y$ is the set of partial functions from X to Y. These are relations which relate each member x of X to at most one member of Y. This member of Y, if it exists, is written f(x). The set $X \fun Y$ is the set of total functions from X to Y. These are partial functions whose domain is the whole of X; they relate each member of X to exactly one member of Y.

The arrows $\pinj$, $\inj$, and $\bij$ with barbed tails make sets of functions that are injective. $(X \pinj Y)$ is the set of partial injections from X to Y. These are partial functions from X to Y which map different elements of their domain to different elements of their range. $X \inj Y$ is the set of total injections from X to Y, the partial injections that are also total functions. The arrows $\psurj$, $\surj$, and $\bij$ with double heads make sets of functions that are subjective. $X \psurj Y$ is the set of partial surjections from X to Y. These are partial functions from X to Y which have the whole of Y as their range. $X \surj Y$ is the set of total surjection's from X to Y, the functions which have the whole of X as their domain and the whole of Y as their range. The set $X \bij Y$ is the set of bijection's from X to Y. These map the elements of X onto the elements of

Y in a one-to-one correspondence. As suggested by its shape, X ⤖Y contains exactly those total functions that are both injective and surjective.

Table 2.6: Brief description about domain and range of functions.

Name	Symbol	Dom f	1-to- 1	Ran f
Total function	$\longrightarrow$	= X		⊆ Y
Partial function	⇸	⊆ X		⊆ Y
Total injection	↣	= X	Yes	⊆ Y
Partial injection	⤔	⊆ X	Yes	⊆ Y
Total surjection	↠	= X		=Y
Partial surjection	⤀	⊆ X		=Y
(Total) Bijection	⤖	= X	Yes	=Y
Finite partial function	⇻	∈ (𝔽 X)		⊆ Y
Finite partial injection	⤕	∈ (𝔽 X)	Yes	⊆ Y

2.2.1.5 Sequences

A sequence is a particular form of function, where the domain elements are all the natural numbers from 1 to the length of the sequence.

Table 2.7: List of sequence notations in Z.

Notation	Name	Definition
seq X	Finite sequences	{f: ℕ ⤀X | ∃ n: ℕ ● dom f = 1 . ..n }
seq_1 X	Non-empty finite sequences	{ f : seq X | #f > 0 }
Iseq X	Injective sequences	Seq X ∩ (ℕ ⤔ X)
〈 a_1,........,a_n 〉	Sequence	{1 ↦ a_1, ... ,n ↦ a_n}
Disjoint e	Disjointness	∀ e_1 , e_2 : dom e |e e. e 1 ∩ e. e_2 = ∅
⁀	Concatenation	⟨a,b,c 〉⁀ ⟨d,,e,f 〉= = ⟨a,b,c,d,e,f 〉
⁀/	DistributedConcatenation	⁀/⟨⟨a,b,c⟩,⟨d,e⟩,⟨f,g⟩⟩==⟨a,b,c,d,e,f,g⟩
↾	Filter	⟨a,b,c,d,e,d,c,b,a 〉↾⟨a,d⟩ = = ⟨a,d ,d ,a⟩.

2.2.1.6 Bags

Table 2.8: List of Bags operators in Z.

Notation	Name	Definition
bag X	Bag	$X \rightarrowtail \mathbb{N}_1$
$n \otimes B$	Bag scaling	$\forall n : \mathbb{N}; B: \text{bag } X; x: X \bullet (n \otimes B) \# x = n^{*} (B \# x)$
$\sqsubseteq$	Bag membership	$\forall x: X; B: B: \text{bag } X \bullet (x \sqsubseteq B \Leftrightarrow x \in \text{dom } B)$
$\sqsubseteq$	Sub-bag relation	$\forall B,C: \text{bag } X \bullet B \sqsubseteq C \Leftrightarrow (\forall x: X \bullet B \# x \leqslant C \# x)$
$\uplus$	Bag union	$\forall B,C: \text{bag } X; x : X \bullet (B \uplus C)\ \ x = B \# x + C \# x$
$\uplus$	Bag difference	$\forall B,C: \text{bag } X; x: X \bullet (B \uplus C)\ \#x = \max\{B\#x - C\# x, 0\}$

A sequence stores information about the multiplicity and ordering of its elements. If wish to record multiplicities, but not ordering, then represent a collection of objects as a bag. Then write ⟦ a, a, b, b, c, c ⟧ to denote the bag containing two copies of a, two copies of b, and two copies of c.

2.2.2 The Schema Language

The mathematical language of Z is powerful enough to describe most aspects of system behaviour. However, the unstructured application of mathematics soon results in descriptions that are difficult to understand. To avoid this, must present mathematical descriptions in a sympathetic fashion, explaining small parts in the simplest possible context, and then showing how to fit the pieces together to make the whole. The process of combining declarations and predicates into structures called schemas. Schemas are used to describe both static and dynamic aspects of a system. The Static Aspects are the state can occupy by the system and invariant relationships that are maintained as the system moves from state to state. Dynamic Aspects like operations that are possible, relationship between their inputs and outputs and change of state that happen during run time of the system.

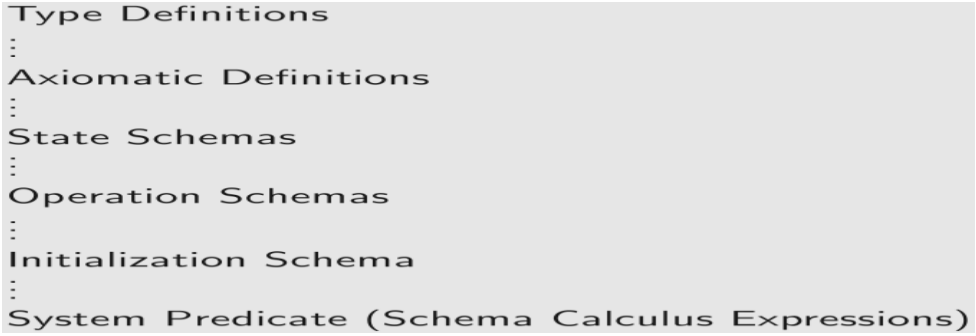

Figure 2.1: Structure of a Z Specification [23].

2.2.2.1 Basic type definitions

[Ident,... ,Ident], Any types which are not defined elsewhere must be declared before the first schema in the above form. A basic type definition introduces one or more basic types. These names must not have a previous global declaration, and their scope extends from the definition to the end of the specification. The names become part of the global vocabulary of basic types. Example **[NAME, DATE].**

2.2.2.2 Axiomatic Descriptions

Var / Const : Type
Predicate; . ..; Predicate

An axiomatic description introduces one or more global variables, and optionally specifies a constraint on their values. These variables must not have a previous global declaration, and their scope extends from their declaration to the end of the specification. The variables become part of the global signature of the specification. The predicates relate the values of the new variables to each other and to the values of variables that have been declared previously, and they become part of the global property. An example of an axiomatic description is the following definition of the function square:

square : $\mathbb{N} \longrightarrow \mathbb{N}$
$\forall$ n :$\mathbb{N}$ $\bullet$ square(n) = n * n.

2.2.2.3 Constraints

Suppose that the state of a system is modelled by a schema State with two components a and b, and that these are introduced under a constraint P.

state

$a : A ; b : B$
p

Each object of schema type State represents a valid state: a binding of a and b in which predicate P is satisfied. We say that P forms part of the state invariant for the system: a constraint that must always be satisfied.

2.2.2.4 Schema definitions

SchemaName

Symbol Declarations
Constraining Predicates

These forms introduce a new schema name. The word heading the box or appearing on the left of the definition sign becomes associated with the schema which is the contents of the box or appears to the right of the definition sign. It must not have appeared previously in the specification, and from this point to the end of the specification it is a schema name. Again, if the predicate part in the vertical form is absent, the default is true. The right-hand side of the horizontal form of schema definition is a schema expression, so new schemas may be defined by combining old ones with the operations of the schema calculus. The vertical form is equivalent to the horizontal form **S ≅ [Symbol Declarations | Constraining Predicates]**.

State schema makes overall statements about the system being specified.

State schema

StateVar : Type
:

Operation schema describes the effect of certain operations which change the state or data in the system by applying pre and post-conditions. Example adding a new customer for the bank, remove the whole queue of vehicles waiting for the ferry.

2.3 Difficulties with Z

- Cannot do concurrency.
- Timing aspects.
- Algorithmic aspects and programming constraints.
- Sequencing operations.

However, there was an admiration that Z wasn't intended to do everything. Remarks here included 'cannot do concurrency in Z, but then it isn't intended for that'. The timing problem with real-time system can be studied under three headings [26]:

System response: The real-time system is to keep performance within promptness requirements. This can be a matter of algorithms, as well as a matter of physical limitations of the underlying hardware.

Event processing: The real-time system is to detect frequency variable in the external environment. This response mechanism of the real time system will depend on the frequency of occurrence of the events, the maximal number of events to be buffered and the worst case average rate.

State processing: The real-time system is to maintain an adequate sampling rate to determine the amplitude variation in the environment. The system can operate in cycles of self- timed or periodic manner.

Time measurement has to do with a numeric notion of time. As most real-time properties hold in specific intervals in a computation, a calculus of such properties can be embedded in an appropriate interval temporal. Several things can be investigated in the context of time measurement such as:

- How should time elements be represented? Should it be mentioned explicitly in the syntax of the language or implicitly in the semantics of the language?
- How should measured be presented? Should it be additive by turning the time domain into a group or metric by adding a distance function like in topology?
- How is time measurement calibrated? Should be it absolute or relative?

As most real-time properties hold in specific intervals in a computation, a calculus of such properties can be embedded in an appropriate interval temporal. Recently, Zhou Chaochen et al [1] shown that many of the requirements of time-critical system can be expressed and reasoned about very effectively in a calculus of durations based on an interval temporal logic.

An interval is a pair $[t_1, t_2]$ such that $t_1, t_2 \in \mathbb{R}$. $[t_1, t_2]$ is a strict interval if $t_1 < t2$. $[t_1, t_2]$ is a non-strict interval if $t_1 \leq t_2$. $[t_1, t_2]$ is not an interval if $t_1 > t_2$. Intervals of the form $[t_1, t_1]$ are called point intervals.

If S is a composite state then $\int S$, the duration of S, denotes a real-valued interval function. For a given interval $[t_1, t_2]$ of a given computation, the value of $\int S$ is

$$\int_{t1}^{t2} S(t)\, dt = \sum_{t1}^{t2} S(t)dt$$

.

The point interval and duration formula are defined as follows:

$$\int_{t1}^{t2} S(t)\, dt = \begin{cases} 0 & \text{if } t1 == t2. \\ l \wedge l > 0 & \text{if } t1 < t2. \\ \text{undefined} & \text{if } t1 > t2. \end{cases}$$

For a given interval $[t_1, t_2]$, the duration formula holds if $t_1 < t_2$ and the value of $\forall\, t \in [t_1, t_2]\bullet$ S (t) =1. Let I be the set of all bounded and closed intervals of real numbers $\{[t_1, t_2]: t_1 \leq t_2 \wedge t_1, t_2 \in \mathbb{R}\}$. l is special temporal variable denoting the interval length and $l([t_1, t_2]) = t_2 - t_1$.

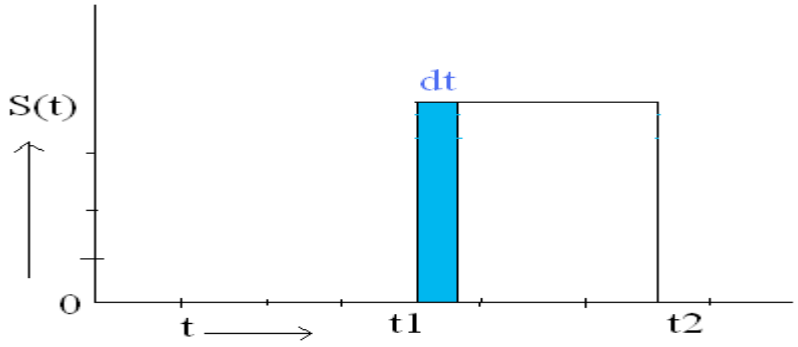

Figure 2.2: Graph of time duration formula.

Distance function maps two time points into a value in metric domain. A **metric** or **distance function** is a function which defines a distance between elements of a set. A

set with a metric is called a metric space. A metric induces a topology on a set but not all topologies can be generated by a metric. A **metric** on a set X is a function (called the *distance function* or simply **distance**) d: $X \times X \rightarrow \mathbf{R}$

(M1) $\forall$ x, y $\in$ X $\Rightarrow$ d(x,y) $\geqslant$ 0. (Non-negative)
(M2) $\forall$ x, y $\in$ X, x = y $\Leftrightarrow$ d(x, y) = 0 (identity of indiscernible).
(M3) $\forall$ x, y $\in$ X, d(x, y) = d(y, x) (symmetry).
(M4) $\forall$ x, y, z $\in$ X, d(x, z) $\leq$ d(x, y) + d(y, z) (sub-additive | triangle inequality).
The discrete metric: if $x = y$ then $d(x,y) = 0$. Otherwise, $d(x,y) = 1$.

In the Euclidean space $\mathbf{R}^n$, the distance between two points is usually given by the Euclidean distance (2-norm distance).

1-norm distance $= \sum_{i=1}^{n} |x_i - y_i|$

2-norm distance $= \left(\sum_{i=1}^{n} |x_i - y_i|^2 \right)^{1/2}$

Captures the abstract state machine underlying a given Z specification as defined in [13]. Using a discrete time model, the variables for time range over the natural numbers $\mathbb{N}$. It is worth mentioning, that instead of $\mathbb{N}$ can be used any infinite, discrete time scale. In [13], that each process has a state space that can be divided into three components: input variables, output variables and internal variables. Working with Z this means that there exist schemas Input, Output and Intern with the declaration of the corresponding variables. The whole state space is then given in Z by

State ≅ Intern ∧ Input ∧ Output. In order to express time dependencies add Time as a fourth component **State ≅ Intern ∧ Input ∧ Output ∧ Time.**

state

Ξ *output*; Ξ *inter*; Δ *input*; Δ *Time*;

t' = t + stutperiod

Definition of duration calculus concepts as they apply to specification in Z is given, this produces a brief notation for expressing temporal requirements in Z specifications.

Chapter 3
Problem Statement

All projects begin with a statement of requirements; it may be difficult to find the "right" definition of a requirement. Determining the characteristics of a given requirement enables the product team to understand nature of a requirement. Correspondingly, allocation of a requirement (or assignment of a relationship to a requirement in a database) enables the product team to visualize and understand the interactive and mutually beneficial nature of balanced requirements set.

3.1 Introduction

A requirement is:

- A statement identifying a capability, physical characteristic, or quality factor that bounds a product or process need for which a solution is to be pursued.
- A clear documentation of the customer's expression of need so as to be directly usable in the design process.

The most common reasons for project failures are not technical. The problems fall into three main categories:

- Requirements – either poorly organized, poorly expressed, weakly related to stakeholders, changing too rapidly or unnecessary; unrealistic expectations.
- Management problems of resources – failure to have enough money, and lack of support or failure to impose proper discipline and planning; many of these arise from poor requirements control.
- Politics– which contributes to the first two problems.

Every Software engineering methodology is based on a recommended development process proceeding through several phases:

>> Analysis, Specification, Design, Coding, Unit Testing, Integration and System Testing, Maintenance.

Quality can be achieved through better and precise understanding of model and implementation, better written software, error detection and correction, test case

generation, and deductive verification. The existence of the formal methods accurse at very early stages of software development process. Formal methods can:

» Be a foundation for describing complex systems in precise way.

» Be a foundation for reasoning about systems.

» Provide support for program development.

Formal methods are system design techniques that use strictly specified mathematical models to built software and hard ware system, uses mathematical proofs as complement to system testing in order to ensure correct behavior. Better understanding of formal methods leads to better software development.

Z is a Formal Specification language that works at a high level of abstraction so that even complex behaviours can be described precisely and concisely, which is also used to specify the functional requirement of the system. The other main ingredient in Z is a way of decomposing a specification into small pieces called schemas. A schema represents a system's state. Schemas are used to describe both static and dynamic aspects of a system. A specification written in Z describes the names of components of the system and expresses the constraints between those components.

3.2 Objectives of Thesis

- Study the importance of formal language Z.
- To find the limitations of Z, which are types of requirements are difficult to specify using the Z language.
- Pay the special attention to finding and correcting errors in the Z specification.
- Overcoming the limitations and correcting the identified errors by using Alloy model (a subset of Z language).
- Implementing the all above objectives using a case study.
- Combining Z with C++ code is also required to convert directly specifications written in Z language to C++ and then verification for the same case study.

Chapter 4
CASE STUDY

An Automated Teller Machine (ATM) is a safety-critical and real time system is taken as a case study. According to requirements of ATM system there are different machine state status, and four different operations: Withdrawal, Deposit, Transfer, and Inquiry. A key part of early design phases are specifications, which span from requirement, to functional, to design, specifications. This chapter describes the conceptual and formal models of the ATM. The formal model of the ATM is specified by using formal specification language. The proper specification language i.e. Z notation, is used to ensure the correctness, reliability and consistency at analysis and design stage, before start the actual implementation of the software system. Z notation is based on set theory and first order predicate logic. A formal model at abstract level has been developed in Z specification language. Z/word tool is used for writing the Z-schemas and the notations. Which support almost all Z symbols are on the Z-Palette. This model is finally checked using Z/EVES toolset.

4.1 Introduction

When developing software systems, the main objective of a requirement specification is to express the user requirements clearly and precisely. The benefits of formal methods and in particular formal specification technique are well documented [12].

A general framework for the specification of concurrent and reactive systems in the formal specification language Z has been presented in [4, 13]. Specification and verification of real-time systems are important research topics that have practical implications. A popular approach for specifying real-time systems relies on the graphical notation Timed Automata [2, 14].

Here ATM system is taken for the application of formal methods. As mentioned earlier ATM is a safety critical system because its incorrect functioning may lead to large scale economic loss. As formal methods are a promising way of giving increased confidence in safety critical systems that is why the system under development is modelled using formal approaches. Due to its safety critical nature many researchers have been contributing in modelling of the system.

Martin Giese et al. [6] build a semi formal model of the withdraw use case of ATM System with the help of UML and OCL. Yingxu Wang et al. [11] presented the formal model of ATM system using Real Time Process Algebra (RTPA) in order to ensure correctness and dependability. Static and dynamic aspects relating to ATM System functionality are covered in this paper.

Maritta Heisel et al. [5] model some aspects of ATM system in Z notation. The model is not the representation of full ATM system and also the safety critical issues have not been addressed in it. This model reflects real world requirements and is the enhancement to the work already presented in [11]. Main problem with informal specification is the inherent ambiguity of textual description. Mathematics can eliminate such ambiguity as it is concise and complex properties can be expressed succinctly. To get the benefits of formal methods, one doesn't need to be mathematically sophisticated. One can use formal methods at any level of sophistication from light weight formal methods to heavy duty formal methods [3]. For the purpose of formal specification, Z notation is used in this thesis for the specification of ATM system. Z is powerful as it can structure large specifications into chunks which can be read and validated easily. Type definition at specification level is also another power full aspect of Z.

4.1.1 Conceptual Model of the ATM

This section describes the conceptual model of the ATM. An Automated Teller Machine is an electronic computerized device that allows customers to directly use a secure method to access their bank accounts, order or make cash withdrawals and check their account balances without the need for a personal cashier. Many ATMs also allow customers to deposit cash or cheques, exchange currency and transfer money between their bank accounts. ATM system is a safety critical system because its malfunctioning may lead someone to great economic loss.

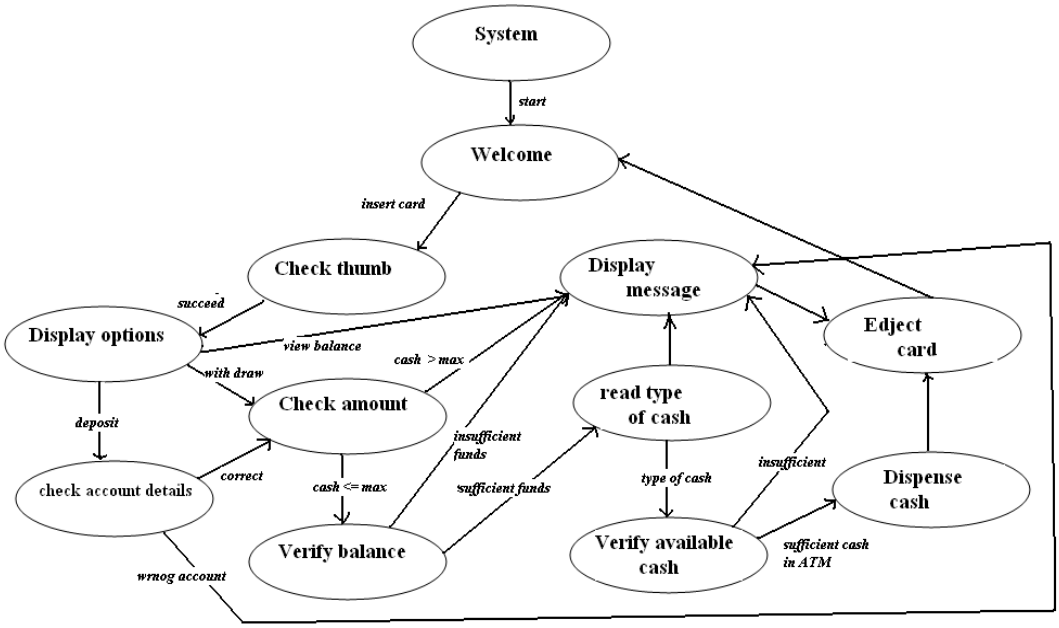

Figure4.1: Conceptual model of ATM.

4.1.2 Requirements Analysis

Banking institution is one of the most important institutions in today's society. The ATM system under consideration provides following services.

1. Card Authorization: The initial step of all customer interactions is to verify the customer's as authorised customer. After this a menu of options is presented on the display which the customer selects by pressing appropriate keys on the keypad. These options lead to other displays and requests for further input.

2. Display account Balance (on screen or printout): Every debit card holder can check his balance.

3. Cash Withdrawal (with or without receipt): The system should allow customers (Debit and Credit customers) to withdraw money from their current accounts.

4. Deposit Money: Every debit card holder can deposit money through ATM.

4.2 Z-specification (Formal model of ATM)

In this section preferred to use an abstract definition, which does not define the necessary data structures, rather than an implementable concrete design. The preconditions and post conditions of each schema, and how the state of the world

evolves are provided within these declarations. If the users of the system call the operations appropriately, unexpected situations cannot happen. Error reports are produced for those situations when the preconditions are not met.

The proposed formal model of ATM system, which is developed by using Z notation. The main advantage of Z notation is that the large specification can be divided in chunks called schemas [8]. By dividing the specification into schemas we can focus and understand the system piece by piece. In Z notation, schemas are used to describe static and dynamic aspects of a system.

4.2.1 Type Declarations

Start of the formalizing process for the system by defining the basic types of the specification. Customer represents whether a credit card holder or debit card holder is while account is a unique account number. Card represents a unique customer card pin number and account number.

⟦Account, Card⟧.

Below are some values that other types can take in the system. T_state stands for an transaction state event name, C_type is a free type for representing currency, T_limit stands for single transaction limit, T_date is transaction date, R_amount is requested amount, and W_amount is defined for the total withdrawal amount in one day. Transaction_ok and Withdrawl_ok are consisting of account number, T_date, R_amount and current balance. REPORT is defined for error handling.

T_state ::= Complete | In Progress | Awaiting For Long | Cancelled.

Action ::= Retain Card | Eject Card | Take Card.

transaction_ok = Account number | Transaction date | Requested Amount | Current Balance.

REPORT ::= ATM Funds Insufficient | Loan Funds Insufficient | Amount Requested Too High | Invalid Card | Invalid Thumb Impression | Have Made Max Withdraw Today | Have Already Made Some Withdrawal Today.

RESPONCE::= success | failure.

RECEIPT::= Balance enquiry | Transaction ok | Withdrawal ok | Deposit.

RESULT::= authentication_ok | prompt for re-enter | transaction ok.

Card ≅ [name: CNAME; pin_no: ℕ; account_id: ℕ; exp_date: Date].

C_Type::= Rupees | Pounds | Dollar | Yen | Mark | Euor | Riyal | Dhiram....etc.

4.2.2 State Specification

Corresponding to the given ATM in Fig 4.1, The ATM system encompasses static behaviours, such as system welcome, verify thumb, check amount, verify account balance, verify cash availability, dispensed cash, and eject card. Each of these static behaviours can be described as a process of the ATM. For instance, the Card can be specified as shown

Card

codedpin : CodedPIN ; cardid :CARDID
lastday :DATE: lastamt AMOUNT ; MaxAmount: AMOUNT

---

lastamt ⩾0
lastamt ⩽MaxAmount
codedpin = ran ℕ

The schema language allows the description of different facets of system. Example, one schema can describe the behaviour of system when it receives the correct input and other schema can describe the system when it receives the wrong input.

The Card Reader (and thumb reader) receives the customer's card and retrieves the PIN and account number stored on it. This information is transmitted to the software system which enables the Customer Keypad and initiates the thumb verification procedure.

Cardreader ::= *boolean* | *T=card inserted* ∨ *f.*

keypadstatus ::= *boolean* | *T=activatedted* ∨ *f = not.*

thumbreaderstate ::= *boolean* | *T = activated* ∨ *f = not.*

CardReader

$card? : \mathbb{P}\ Card$

$cardno : \mathbb{N}$; T :*Cardreader* ; T: *keypadstatus*

T : *thumbreaderstate*

---

syatem.Cardreader = $T \Rightarrow$ *system.keypadstatuse* = $T \wedge$

system. thumbreaderstate = T

Starting from the authentication process ok, ATM reads the card and thumb details from the user, and checks them whether they are available in the database. For this card authentication schema uses the other schema details of Card Data Base. Every card presented in Card Data Base should Has a one-one relation with bank Account.

CardDataBase

Known_card : $\mathbb{P}$ *Card*

Cardno : $\mathbb{N}$ (*seq Card*) ; *Has* : *Card* $\rightarrowtail\!\!\!\!\!\rightarrow$ *Account*

Known_card = dom(*Has*)

AlreadyKnownCard

Ξ *CardDataBase*

Result! : *RESULT* ; *Card*? : *Card*

Has : *Card* $\rightarrowtail\!\!\!\!\!\rightarrow$ *Account*

Card? $\in$ *Known_Card* [*or dom* (*Has*)]

Result! = *ok* | *already_known* | *welcome*

If inputted card is existing in the Known Card list then the result is Ok or displays the welcome screen. The next is Thumb authentication process. Similarly, Thumb authentication schema uses the other schema details of Thumb Data Base. Every Thumb presented in Thumb Data Base should Has a one-one relation with Card.

Count : $\mathbb{N}$

$\forall$ *card*? $\in$ *Has*; *card*?$\bullet$*Count* = 0

ThumbDatabase

$Already present : \mathbb{P} ThumbImpression$

$Thumb : seq\ ThumImpression$

$Has1 : Thumb \rightarrowtail\!\!\!\!\!\!\rightarrow Card$

$Already present = dom(Has1)$

ThumbAuthentication

$\Xi\ ThumbDatabase$

$Result! : RESULT ; Count, Count' : \mathbb{N}$

$Thumb? : ThumbImpression ; Has1 : Thumb \rightarrowtail\!\!\!\!\!\!\rightarrow Card$

$Thumb? \in Already present\ [or\ dom\ (Has1)] \wedge Count \leqslant 3$

$Result! = ok\ or\ already\_known\ or\ welcome \wedge Count' = 0$

Bank: Bank maintains the details of customer, account type, ATM critical amount value, account critical value, loan critical value and status of the customer. Based on the account type, account critical value, loan critical value and ATM critical level, the bank maintains the status of the customer. There are only two values adequate and inadequate for fund status which is declared as a free type.

For every card in data base the withdrawal limit from ATM is Rs.60, 000 per day and can be withdrawal or deposit the amount of 20,000 in single transaction. Initially for all cards the withdrawal amount is set be zero for each day. In bank schema D_C_Holder and C_C_Holder stands debit card holder and credit card holder respectively. C_card is Current Card, A_Amount is Available amount, and A_balance is account balance.

Bank[*AccountId,Customer*]

$Cards : \mathbb{P}\ Card$

$Accounts : \mathbb{P}\ Account$

$Has : Card \leftrightarrow Account$; $Has3 : Account \mapsto AccountType$

$C\_Card? : \mathbb{P} Card$

$A\_balance , T\_limt, A\_Amount, Max\_deposit , W\_amount : \mathbb{N}$

$Acc\_critical\_value , Loan\_critical\_value , Atm\_critical\_level : \mathbb{N}$

$AtmFunds,LoanFunds,LoanFunds' : \mathbb{N}$; $FundStatus ::= adeque \vee inadeque$

$Today? : Date$; $Accid? : Accoun$; $Acctype? : AccountType$

$D\_C\_Holder, C\_C\_Holder : Customer$

$\forall Account : (Accid?, Acctype?) \in Has3 \wedge Accid? \bullet A\_Balance \geqslant Acc\_critical\_level.$

$C\_C\_Holder(Accid?,Acctype?) \bullet FundStatus=adequate \Leftrightarrow$

$(AtmFunds \wedge Atm\_critical\_level) \wedge (LoanFunds \wedge Loan\_critical\_value).$

$Accid?(Accid?, Acctype?) \bullet FundStatus=adequate \Leftrightarrow$

$(AtmFunds \wedge Atm\_critical\_level) \wedge (AccountBalance \wedge Acc\_critical\_value).$

$C\_Card? \bullet T\_limit = 20000$; $\forall C\_Card? \in dom(Has).$

$Today? \bullet C\_Card? \bullet AvailableAmountt = 60000$; $\forall C\_Card? \in dom(Has).$

$Today? \bullet C\_Card? \bullet W\_amount = 0$; $\forall C\_Card? \in dom(Has).$

Automated Teller Machine (ATM): ATM has two states Active or Inactive. ATM reads the current card, thumb, amount from the user, and checks the maximum withdrawn from the account. And also allows the user to withdraw various types of currency. The maximum with drawl for an account throw ATM is Rs 60,000. When card is inserted, ATM generates a serial number and uses as a receipt number. If two cards have the same number then both are same cards. If the account balance of current customer is less than the account critical value then ATM restrict that customer and don't allow retrieving the amount. The system requires that the Card Reader is able to receive the following commands:

ENABLE: Makes the Card Reader ready to receive a card.

DISABLE: Prevents the Card Reader from accepting a card.

When a card is detected in the Card Reader, Input is received initially from the Card Reader and then directly from the customer via the Customer Keypad. The customer receives output from the Customer Display, the Printout Dispenser and the Cash Dispenser. The initial step of all customer interactions is to verify the authentication. After this a menu of options is presented on the display which the customer selects by pressing appropriate keys on the keypad. These options lead to other displays and requests for further input. Some options require account details which are retrieved from the Accounts Database and may also involve updating the database. During the final stage of all customer interactions the Card Reader is instructed to either return or confiscate the card. The ATM main schema is

ATM

$\Delta Bank$

$Status ::= \{boolean \mid T = Atmactive \lor F = customermode\}$

$Cardreaderstatus ::= \{boolean \mid T = enable \lor F = disable\}$

$CardEjectstatus ::= \{boolean \mid T = Eject \lor F = NoAction\}$

$Sr\_no, Acc\_critical\_value, Balance : seq \mathbb{N}$; $C\_Card? : Card$

$today? : Date$; $T\_limit : \mathbb{N}$; $Retrivies ::= boolean$

$Restrict ::= \{boolean \mid T = DontAllow \lor F = Allow\}$

$C : Card$; $C \bullet Sr\_no = ran\, \mathbb{N}$

$System.status = T \Rightarrow system.Cardreaderstatus = T$

$System.status = F \Rightarrow system.Cardreaderstatus = F$

$C\_Card? \bullet Restrict = F \Rightarrow Atm \bullet status = T$

$C\_Card? \bullet Balance = Acc\_critical\_value \Rightarrow C\_Card? \bullet Retries = 0$

$C\_Card? \bullet Restrict = T \Rightarrow C\_Card? \bullet Retries = 0$

$C1, C2 : Card$; $C1 \bullet Sr\_no == C2 \bullet Sr\_no \Rightarrow C1 == C2$

4.2.3 Error Handling

Errors that break the constraints of the system, below are some schemas that report errors based on whether the known card, or the transaction limit, or the sufficient amount funds. Note also that they do not change the state but simply quit the application with an appropriate message.

When the customer chooses a transaction type from a menu of options, customer will be asked to furnish appropriate details (e.g. account(s) involved, amount). The transaction will be sent to the bank, along with information from the customer's card and the customer thumb details.

If the bank reports that the customer's details are invalid, the Invalid extension will be performed and then an attempt will be made to continue the transaction. If the customer's card is retained due to too many invalid trials, the transaction will be aborted, and the customer will not be offered the option of doing another.

UnKnownCard

Ξ *CardDataBase*

card? : *Card* ; *report*! : *REPORT*

Has : *Card* $\leftrightarrow$ *Account*

---

card? $\notin$ *dom* (*Has*) $\Rightarrow$

report! = *" invalid card "* | *" un known card"*

Invalidate

Ξ *CardDataBase*

Ξ*ThumbDataBase* ; *card*? : *Card*

thumb? : *ThumbImpression*

report! : *REPORT*

---

card? $\notin$ *dom* (*Has*) $\vee$ *thumb*? $\notin$ (*Has*1) $\Rightarrow$

report! = *" invalid card "* | *" un known thumb impression "*

After the authentication the system will displays the various options like balance enquiry, withdraw amount, deposit amount and mini statement ....etc. For withdraw the system will checks the amount. If the amount is large or currency type is less than the requested amount then the system should generates the amount is too high. The below schema displays the report "Requested amount is too high".

AmountTooHigh

ΞATM

$R\_Amount, A\_Amount : \mathbb{N}$; $Card? : \mathbb{P}Card$

$Type : C\_Type$; $Report! : REPORT$

$Card?\bullet R\_Amount > T\_limit \lor ((type * R\_Amount) + W\_Amount) \geqslant$

$Card?\bullet A\_Amount \Rightarrow$

$Report! = Requested\ Amount\ is\ Too\ High$

Some time the requested amount may be more than the amount available in ATM or the requested type of currency may not available or ATM funds may not available. Such cases ATM generate the insufficient funds report.

InsufficientATMfunds

$\Xi Bank$

ΞATM

$R\_amount , A\_Amount: \mathbb{N}$

$Type : C\_Type$; $card? : Card$

$report! : REPORT$

$card? \in dom\ (Has1); card?\bullet R\_amount \geqslant ATM\bullet A\_Amount \lor Type \notin C\_Type$

$\Rightarrow Report! = Insufficient\ Amount$

The schema by taking the today date and account number verifies whether transaction limit is exceeded or equal to the withdrawal amount. If that is the case then system displays the message as maximum transaction has been made for today. Otherwise it adds the requested amount to the maximum withdrawal and generates the total withdrawal amount report. This schema uses another three schemas named ATM, Bank and TransactionID.

TransactionLimitExceed

ΞATM

$\Xi Bank$

$\Xi TransactionID$

$Today?, T\_Date : Date$

$T\_Limit , R\_amount , AvailableAmount , W\_amount , W\_amount' : \mathbb{N}$

$card? : Card ; Thisaccount? : Account$

$Report! : REPORT$

$If [\![(Today? = = T\_Date) \wedge$

$(Today?\bullet (card?, Thisaccount?)\bullet W\_amount = = (card?, Thisaccount?)\bullet$

$A\_Amount]\!] \Rightarrow Report! = \text{“Already made the maximum withdraw for today.”}$

$If [\![Today?\bullet (card?, Thisaccount?)\bullet W\_amount + R\_amount \leqslant$

$(card?, Thisaccount?)\bullet A\_Amount]\!] \Rightarrow$

$[\![Today?\bullet(card?,Thisaccount?)\bullet W\_amount = Today?\bullet(card?,Thisaccount?)\bullet$

$W\_amount + Today?\bullet (card?, Thisaccount?)\bullet R\_amount]\!]$

4.2.4 Event Handling

If requested amount is sufficient then ATM generates the transaction report. Transaction of ATM includes transaction date, account number, requested amount, and pin_no. Transaction date is of type date, R_amount is of type natural number, account no is of type Account Id which is already defined. Transaction Id schema is

TransactionId

$\Xi Bank ; \Xi ATM$

$T\_date : Date ; R\_amount,Balance, Pin\_no : \mathbb{N}$

$Report! : RESULT ; card? : Card$

$Result = Transaction\_ok$

Whenever a transaction occurs, its account number, transaction date, pin_no, and transaction amount should be stored in TransactionId schema.

The balance enquiry schema includes serial number, T_date, serial number and current balance.

BalanceEnquery

$\Xi Bank$

$Thumb? : ThumbImpression$

$Balance : \mathbb{R}$

$Sr\_no : \mathbb{N}$

$Today? : Date ; Card? : Card$

$Report! : REPORT$

$Receipt! : RECEIPT$

$Card? \in dom(Has)$

$Result! = transaction$

$Receipt! = Sr\_no$

$Receipt! = Balance\ Enquiry$

A withdrawal transaction asks the customer to choose a type of account to withdraw from (e.g. checking) a menu of possible accounts, and to choose a dollar amount from a menu of possible amounts. The system verifies that it has sufficient money on hand to satisfy the request before sending the transaction to the bank. (If not, the customer is informed and asked to enter a different amount.). If the transaction is approved by the bank, the appropriate amount of cash is dispensed by the machine before it issues a receipt. (The dispensing of cash is also recorded in the ATM's log.)

A withdrawal transaction can be cancelled by the customer pressing the Cancel key any time prior to choosing the dollar amount.

WithDraw

$\Delta Bank$; ΔATM

$\Xi ThumbDataBase$; $\Xi CardDataBase$

$C\_Card? : Card$; $AccId? : AccountID$

$Thumb? : ThumbImpression$; $Today? , T\_date? : Date$

$R\_Amount? , W\_amount , W\_amount' : \mathbb{N}$

$Balance, Balnce' : \mathbb{R}$

$Result! : RESULT$; $Responce! : RESPONCE$; $Receipt! : RECEIPT$

$\{ ((Card? , Thumb?) \in Has1) \land ((C\_Card? \bullet R\_Amount) \leqslant (C\_Card? \bullet T\_limit))$

$\land ((Today? \bullet C\_Card? \bullet W\_Amount + C\_Card? \bullet R\_amount)$

$\leqslant (C\_Card? \bullet A\_Amount))$

$\land ((C\_Card? \bullet R\_Amount) \leqslant (C\_Card? \bullet Balance)) \land$

$((C\_Card? \bullet R\_Amount) \leqslant (ATM \bullet Balance)) \}$

$\Rightarrow \{ ((AccId(Card? , Thumb?) \bullet Balance') =$

$(AccId(Card? , Thumb?) \bullet Balance) - (C\_Card? \bullet R\_Amount)) \land$

$((AccId(Card? , Thumb?) \bullet W\_amount') =$

$(AccId(Card? , Thumb?) \bullet W\_amount) + (C\_Card? \bullet R\_Amount))$

$((ATM \bullet Balance') = (ATM \bullet Balance) - (CurrenCard? \bullet R\_Amount)) \}$

$Responce! = Success.$

$T\_date? = Today?$

$Result! = Transfer\_ok$

$Report! = WithDrawl\_ok$

A deposit transaction asks the customer to choose a type of account to deposit to (e.g. checking) from a menu of possible accounts, and to type in a dollar amount on the keyboard. The transaction is initially sent to the bank to verify that the ATM can accept a deposit from this customer to this account. If the transaction is approved, the machine accepts an envelope from the customer containing cash and/or checks before it issues a receipt. Once the envelope has been received, a second message is sent to the bank, to confirm that the bank can credit the customer's account contingent on

manual verification of the deposit envelope contents by an operator later. (The receipt of an envelope is also recorded in the ATM's log.) .

A deposit transaction can be cancelled by the customer pressing the Cancel key any time prior to inserting the envelope containing the deposit. The transaction is automatically cancelled if the customer fails to insert the envelope containing the deposit within a reasonable period of time after being asked to do so.

Deposit

$\Delta Bank$; ΞATM; $\Xi ThumbDataBase$; $\Xi CardDataBase$

$To\_AccountNO? : AccountID$

$Acctype? , To\_acctype? : AccountType$

$Name? : CustomerID$; $C\_Card? : Card$

$Thumb? : ThumbImpression$; $HAS4 : AccounID \leftrightarrow CutomerID$

$balance , balance' , R\_amount? , A\_Amount , T\_limit : \mathbb{N}$

$Report! : REPORT$; $Result! : RESULT$; $Receipt! : RECEIPT$

$\{ C\_Card? \in (Has) \land C\_Card?\bullet Acctype? = = debit\ card\ holder \land$

$C\_Card?\bullet balance (C\_Card? , Acctype?) > R\_amount? \land$

$C\_Card?\bullet R\_amount + Today?\bullet C\_Card?\bullet W\_amount$

$\leqslant C\_Card?\bullet A\_Amount \land$

$C\_Card?\bullet R\_amount \leqslant C\_Card?\bullet T\_limit \}$

$\Rightarrow$

$\{ C\_Card?\bullet balance' (card? , Account) =$

$Currentc\_Card?\bullet balance (C\_Card? , Acctype?) \oplus$

$\{ C\_Card?\bullet balance (C\_Card?, Thumb?) - C\_Card?\bullet R\_amount \}$

$To\_AccountNO?\bullet balance' (To\_AccountNO? , To\_acctype?) =$

$To\_AccountNO?\bullet balance \oplus$

$\{ (To\_AccountNO? , Name?) \mapsto To\_AccountNO? \bullet balance +$

$C\_Card?\bullet R\_amount \}$

$C\_Card?(Card? , Thumb?)\bullet W\_amount') =$

$(C\_Card? (Card? , Thumb?)\bullet W\_amount) + (C\_Card?\bullet R\_Amount) \}$

4.3 Verification

To type check a document clicks the fuzz button in Z/Word tool. The results are displayed in a dialog box.

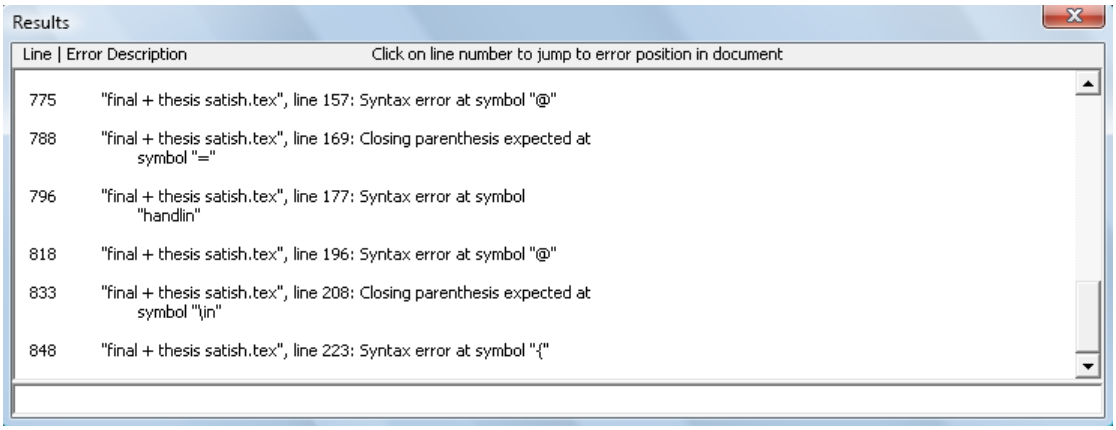

Figure 4.2: Type checking the Z specifications.

After type checking Z/Word tool generates a .tex file, which is used as input for the Z/Eves tool.

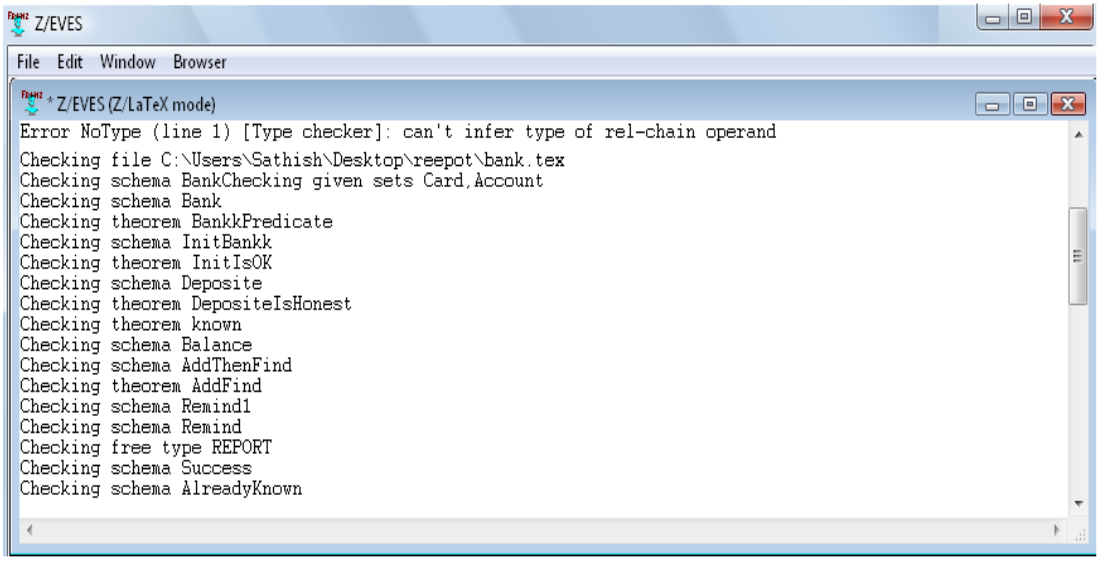

Figure 4.3: Verification of Z specification.

Special attention is paid to finding and correcting errors in the Z specification that we are modelling and bugs in the Alloy model itself [chap 5]. The following specifications are difficult to model in the Z language:

1. Pin number measurement or any constant is not modelled.

2. Invalid Thumb Extension.

3. The maximum number of trials for invalid thumb extension is 3.

4. Add thumb and thumb authentication should be inherited from the thumb data base.

5. Allow transaction should be combination of both card and thumb authentication processes.

6. NO card should be both credit and debit.

7. Every card presented in the data base should be mapped with at most one thumb.

8. The card authentication schema should extend card data base schema and thumb authentication schema should extend thumb database, and both should extend bank schema.

9. When authentication succeeded the menu of option is displayed. If customer selected one option then it should not match with other.

10. Each Deposit, Withdrawal and Balance enquiries are should be extension of ATM.

These specifications are represented using an Alloy Model for ATM and are discussed in next chapter.

Chapter 5

Alloy Model

5.1 Motivation For Using Alloy

Alloy is a formal language deeply rooted in Z. It was developed by Daniel Jackson, currently at MIT. A recently developed notation, strongly influenced by Z and similar to the OCL notation, is Alloy. Alloy is an ASCII-based first order logic modelling language supported by a tool for fully automated syntactic and semantic analysis. For a full introduction to the language syntax and its semantics refer to [27].

How is Alloy related to Z?

Z was a major influence on Alloy. Very roughly, Alloy can be viewed as a subset of Z. Unlike Z, Alloy is first order, which makes it analyzable (but also less expressive). Alloy's writing mechanisms are designed to have the flexibility of Z's schema calculus, but are based on different idioms: extension by addition of fields, similar to inheritance in an object-oriented language, and reuse of formulas by explicit parameterization, similar to functions in a functional programming language. Alloy is a pure ASCII notation and doesn't require special typesetting tools.

This section aims to explain why the Alloy language was chosen to be part of the proposed approach. It presents Alloy's main features and some basic knowledge on Alloy, required for understanding the further abstraction analysis.

5.1.1 Alloy Main Features

Alloy [25] is a modelling language based on first order logic, used for expressing complex structural constraints and behaviour. With the underlying idea that code is a poor medium for exploring abstractions , the Alloy language provides declarative high level descriptions (or models) allied with lightweight formalism.

The Alloy Analyzer is a constraint solver that provides fully automatic simulation and checking of Alloy models. It translates constraints to be solved from Alloy into Boolean constraints, which are then fed to an off-the-shelf SAT solver. All problems are solved within a user specified scope that bounds the size of the domains, and thus renders the problem finite (and reducible to a Boolean formula).

What makes Alloy distinct from other languages used for specification is that models in Alloy are both declarative, analyzable, support complex structural states (instead of complexity due to event sequencing) and are usually much smaller than their respective implementations. None of these features is new in themselves. Formal specifications in Z are declarative and structural, and model checking languages such as SMV are analyzable [25].

5.2 Basic Notions

This chapter introduces the mathematical constructing blocks in Alloy. In comparison to a standard presentation of first order logic and basic set theory.

Alloy = logic + language + analysis.

The Logic: At the core of Alloy is a relational logic that combines relational algebra with first order predicate logic. Structures are composed of atoms and relations. Atoms represent typed, immutable structures that are un interpreted and can be related through relations. A relation is a set of tuples each being an atom and can have arbitrary arity. Relations are combined with operators to form expressions.

The Language: In addition to the logic, Alloy provides some language constructs to help organize a model. A model in Alloy may consist of signatures (sig), facts (fact), functions (fun), predicates (pred), and assertions (assert).

The Analysis: The Alloy Analyzer (AA) [35] is an automated tool for analyzing models written in Alloy. Two kinds of analysis are enabled by AA, based on commands. The first is simulation (using run command) whereby the validity of a predicate or function is verified by showing a snapshot of the system for which the predicate is valid. The second analysis technique is checking (using check command), where by an assertion is tested and AA tries to find a counter example.

5.2.1 Logic: Set, Unary, Boolean, Cardinality Operators and Quantifiers

In Alloy, every expression denotes a relation. Relations can be of any arity. Sets of atoms are represented by unary relations and scalars by singleton unary relations. So depending on the type and arity of the given arguments, the operators will yield a (singleton) unary, binary or n-ary relation.

Table 5.1: The logic set operators in Alloy.

Set Operator	Meaning
+	Union
&	Intersection
-	Difference
=	Equality
in	Membership
->	Cross product

The standard quantifiers are:

all x: e | F // universal: F is true for every x in e.

some x: e | F // existential: F is true for some x in e.

all disj x, y: e | F says that F holds whenever x and y are given different values drawn from e.

Table 5.2: List of Boolean operators.

Boolean operators	Meaning
!F	negation: not F
F && G	conjunction: F and G
F || G	disjunction: F or G
F => G	implication: F implies G
,	for alternative, or separation
F <=> G	bi-implication: F when G

In addition, Alloy provides the following quantifiers:

no x: e | F // F is true for no x in e.

sole x: e | F // F is true for at most one x in e.

one x: e | F // F is true for exactly one x in e.

The basic format for a quantifier is **quantifier** variable: type | formula. Where the formula may includes references to the quantifier variable.

Multiple variables can be used in the same quantifier, like:
quantifier variable: type, variable': type' | formula.

Examples: *every file is in some directory*.

all f:File | **some** d:Dir | f.parent = d.

Table 5.3: Unary operators.

Unary operators	Meaning
→	for product
. , []	dot join, box join
~	for transpose
^	for transitive closure
*	for reflexive transitive closure

5.2.2 Language: Module, Signatures, Facts, Functions, Predicates, and Assertions

5.2.2.1 Signature

A declaration of the form **sig** S {…} introduces a signature S, consisting of a basic type and a set of atoms drawn from that type. An unspecified type (apart from integer) becomes an empty signature. For example, **sig** person {} introduces person as a un interpreted type (or a set of indivisible atoms). The declaration **sig** S {f: E} **sig** E {} declares a basic type S and a relation f of type E. If some x has the type S, the expression **x.f** will have the type E.

Signatures may be extended by using the extend keyword. A signature declared as an extension is a sub signature and creates only a set constant along with a constraint making it as a subset of each super signature listed in the extension clause. The sub signature takes on the type of the super signature. However, a sub signature can't declare a field whose name is the same as the name of a field of its super signature.

Thus a sub signature can't override the fields of a super signature. For example, **sig** man, women **extends** person {}.

Abstract signature has no elements except those belonging to its extensions. For example:

abstract sig A { f: set B } {--constraints go here}

abstract sig B {}; **sig** A1 **extends** A {}; **sig** A2 extends A {}

An enumerated (free) type becomes signatures and its elements extend it: Colour == Red | Green | Blue.

sig Colour {} ; **sig** Blue, Red, Yellow extends Colour {}.

5.2.2.2 Facts

A fact is simply a constraint that is assumed always to hold, and hence needs not be explicitly invoked. Facts usually describe global model constraints. The facts and the signature constraints thus constitute a complete set of structural constraints over the model. Syntax for *fact* statements

fact [name] { [list of constraints]

[Multiple constraints are implicitly conjoined]}.

Facts do not need to have names, and are never referred to by name. However, good names improve clarity tremendously. For example

sig Person {name: Name, pets: **set** Pet}

sig Pet {name: Name}

fact {**all** p: Person | **no** p.name & p.pets.name}

In the body of the fact, the first occurrence of the identifier name refers to the name relation of Person, and the second refers to the name relation of Pet. It says that persons don't share names with their pets.

When Alloy searches for examples, it discards any which violate any **fact**. Thus, if the fact is trivially false, then you will simply get no examples of file systems, even if the

assertion you are checking has counterexamples. It will appear the same as if the model were bug free and the assertion you are checking is correct [35].

5.2.2.3 Functions

A function, declared with **fun**, is a named reusable expression that can be invoked within the model. A function takes zero or more arguments and returns either a true/false or a relational value. A function is just a binary relation with the property that it maps each atom in the left set to at most one atom in the right set.

A function (fun) is a parameterized formula that can be "invoked" elsewhere. For example, the function f declared as:

fun f(p_1: T_1, ..., p_n: T_n) { ... }

have n parameters: p_1, ..., p_n of types T_1, ..., T_n respectively.

all result: D | f (a_1, result, a_2, …, a_n) => F [result/e]

Where D is the right-hand side of the declaration of the (second) missing argument, and F [result/e] is F with the fresh variable result substituted for the application expression E. The application of f in this elaborated formula is now a formula, and is treated simply as an in lining of the formula off.

5.2.2.4 Module Header

Alloy models are divided into modules. The first non-comment of an Alloy model is module. Every Alloy model must begin with a module declaration.
module path/model Name.

The module statement specifies the location of the package that the model belongs to. The path is relative to the "models" directory in the Alloy distribution. The text of the model should be in a file named "modelName.als" at the location specified by the path.

It is possible to put entire model in a single module. Some models or model fragments are used repeatedly in other models. To make reuse convenient, and to allow structuring of large models, Alloy allows a model to incorporate the contents of other modules. Alloy's module system is a simplified version of Java's package system.

Each element in a module–that is, signature, fact, function or assertion–has a qualified name obtained by prefixing the element's name with the module name. To refer to an element in the same module, use the qualified name, or just the element's declared name.

A model named **ord** can be imported into any other model by writing

open std / ord.

In this case, **std/ord** is the path where **ord** is stored, relative to the "models" directory in the Alloy distribution. **std** is a folder containing standard packages provided with Alloy.

This is equivalent to preopening the contents of the **ord** model to any model. Consequently, careful about name conflicts. In general, to write entire model in one module (one file), **Open** is primarily for importing utilities, such as the **ord** module (which provides a way to do ordered state). To be able to make reference at all to an element from a different module, must import the module explicitly at the top of the file. An **import** declaration with the key word used to makes the elements of the module available; and additionally, allows them to be referred to by their unqualified names.

5.2.2.5 Predicates

A predicate, declared with **pred**, is a named reusable constraint that can be invoked. A predicate takes zero or more arguments. The named formula with declaration parameters are predicates that return a value of "acceptable" or "not acceptable". Syntax for pred (predicate) statements

pred name (parameter1, parameter2) {

[List of constraints -- each must evaluate to true or false]

[Multiple constraints are implicitly conjoined]}.

If the inputs satisfy all of the constraints listed in the body, then the predicate evaluates to true. Otherwise it evaluates to false. One reflection of this is that predicates in Alloy only ever evaluate to *true* or *false*, while function may evaluate to relations.

5.2.2.6 Assertions

Assertions, declared with assert, is a named constraint that is intended to follow from the model's facts. Assertions take no arguments and are usually checked by the Alloy Analyzer. Assertions constraint intended to follow from facts of the model. While a **fact** is used to *force* something to be true of the model, an **assert** is a claim that something must already be true due to the rest of the model. Assertions are verified (or falsified) using a check statement.

assert name-of-assertion { //list of constraints }.

Example: **assert** acyclic {no d: Dir | d in d.^contents }.

Assertions *must* be names (unlike facts), since they will need to refer to them by name in **check** statements.

5.2.3 The Analysis: Check and Run Commands

5.2.3.1 Check command

With Alloy, automatically verify (up to a specified scope) or falsify (with a counter example) an assert statement, by using a **check** command.

check name-of-assert for **integer.**

It is possible to check only one assertion at a time. To check multiple assertions at once, write and check a new assertion that combines them. The integer defines the scope of the maximum size of each set in the model that will be considered. Specifically, it is the maximum size of the top-level signatures (sets).

Larger scopes take longer to solve, and often result in more complicated examples. Thus it is good practice to reduce the scope as low as it can go but still get a solution, before trying to visualize solutions.

check acyclic for 5 **but** 1 file System, 7 Object. Alloy will examine all file systems with up to 5 **Object**s, and try to find one that violates the **acyclic** assertion. In general, saying "**for 5**", means that Alloy will examine all examples whose top level signatures (those that don't **`extend`** other signatures) have up to 5 instances. The **but** command specify exceptions to the main scope. In this case, Alloy will only examine 1 file system, up to 7 file system objects, and up to 5 of any other set.

5.2.3.2 Run command

The **run** command generates sample solutions to the given model.

run pred-name for # but # sig-name, exactly # sig-name

run pred-name for # sig-name, sig-name, sig-name, # sig-name.

If no default is given (the first # in the first of the two templates shown above, then it is assumed to be 3). One benefit of using the **run** is that it will increase the confidence that the model is not under constrained. Doing runs to periodically check up on system to helps identify over constrained; if Alloy is unable to find *any* solutions when it should have, then the model is surely over constrained.

5.3 Converting the Z specification into Alloy

Table 5.4: Converting the Z specification into Alloy.

Z specification of a Bank	**Alloy specification of a Bank**
Basic types of the Z specification: *[CUSTOMER, ACCOUNTS].* Schema for state space of the system, Customers is the set of Customer name, Accounts is the set of account numbers and Has is a function which, when applied to certain names, gives the account is associated with them *Bank* $Customers : \mathbb{P}\ Customer$ $Accounts : \mathbb{P}\ Account$ $Has : Customer \leftrightarrow Account$ $Customers = dom\ Has$ Operation to deposit money, the declaration ΔBank (account) describes	Basic Alloy signatures: *sig CUSTOMER{}* *sig ACCOUNT {}.* A bank has three fields: Customer, a set of customer names, Account is the set of accounts and Has, a function from Customer to Accounts: *sig Bank {* *Customer: set Customer* *Accounts : set Account* *Has: Customer -> Account* *}.* The operation Deposit adds the

a state change. The Deposit function adds the amount to the customer account if there is a map between customer and account number and displays transaction ok. *Deposit* $\Delta Bank(account)$ $amt?, acc?, balance, balance' : \mathbb{N}$ $m! : Report$ $\exists c : customer, a : Account \bullet$ $c.number = acc? \land (c, a) \in Has \land$ $a.balance' = a.balance \oplus$ $\{(c, Acc?) \mapsto Acc? \bullet balance + amt?$ $\land m! = transaction\_ok.$	amount to the customer account if there is an association between customer and account number. The ' symbol indicates a state change for the Bank account: ***Fun*** *Deposit* *(c: Customer, a: Account, amt,bal: Int)* *{* *acc.bal' = acc.bal++* *{acc.bal(c->a) add amt }* *}*

This section describes steps to convert Z specification into an Alloy model. Goal in writing this model is to describe some aspects of the system (but not the entire system), to constrain it to exclude ill-formed examples, and automatically check properties about it. A special attention is given to finding and correcting errors in the Z specification that are modelling and bugs in the Alloy model itself. Proposed Alloy model directly followed Z specification with the following medications:

1. **NO card should be both credit and debit.**

module Unique_Card ; sig Card, Account, Customer {}

sig Bank { card : set Card; account : set Accounts; bal, acc_critical_val : ℕ

Has: Card one -> Account one

loan_critical_value, loan funds, atm_funds, atm_critical_value : ℕ

debit_card_holder, credit_card_holder : Card }.

sig credit_card_holder, debit_card_holder extends Card {}

fact { **all** t : Card | no credit_card_holder.t & debit_card_holder.t }.

run {**some** Bank} expect 8.

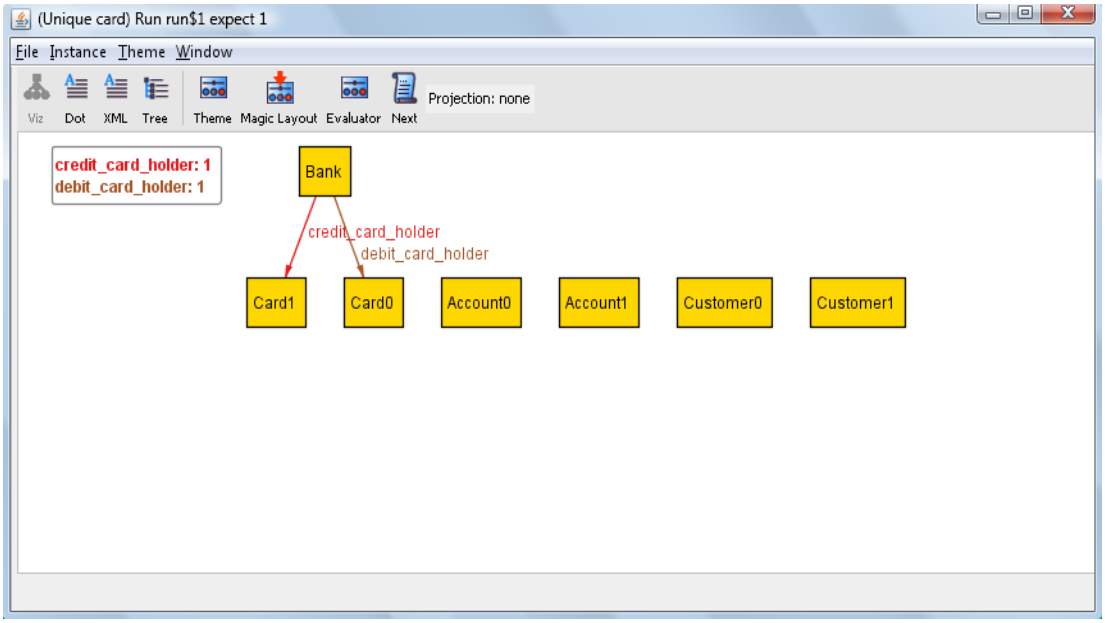

Figure 5.1: Visualization view of NO card should be both credit and debit.

In diagram produced by the Alloy Analyzer shown in Figure 5.1, card0 and card1 are two cards having different accounts and customers. The above figure 5.1 shows that they are two different types debit card and credit card with different accounts. This visualization solution is generated by the Alloy Analyzer. It also provides deferent views like document view, XML view and Tree view. By clicking on next button it will generates the new solution as shown in below figure

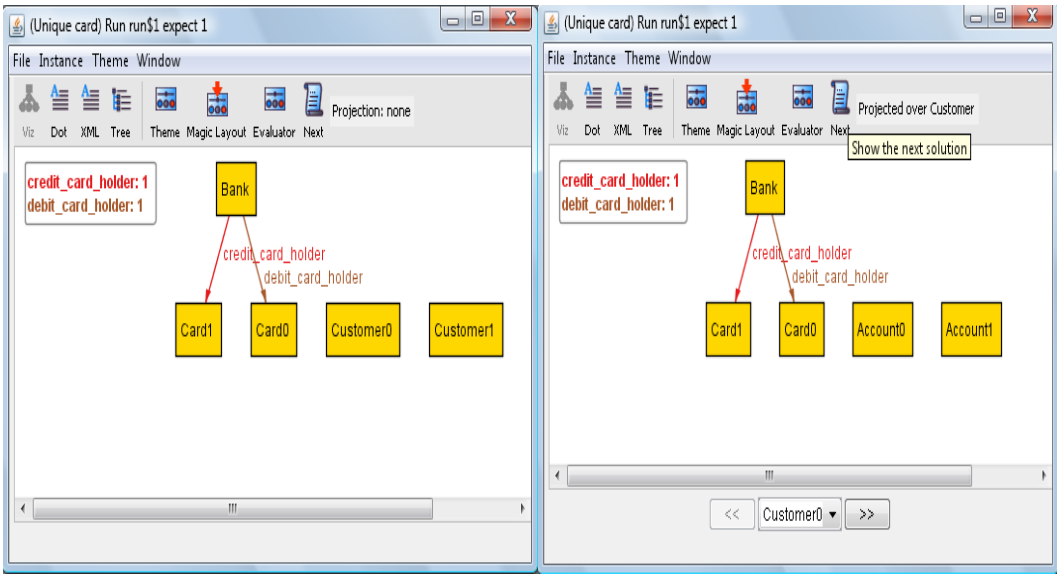

Figure 5.2: Next solution of NO card should be both credit and debit.

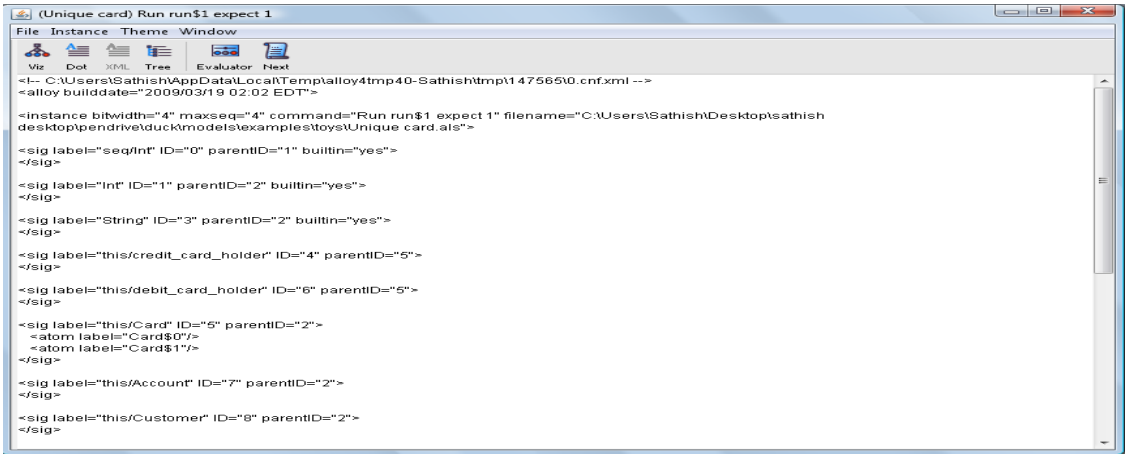

Figure 5.3: XML view of NO card should be both credit and debit.

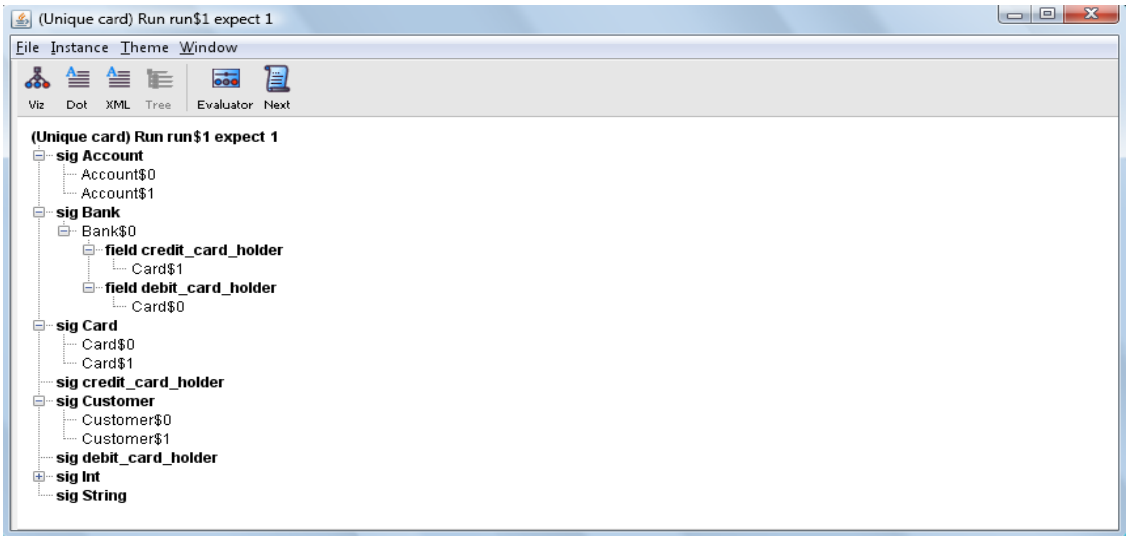

Figure 5.4: Tree view of NO card should be both credit and debit.

For every customer there are only two types of possible based on some conditions. They have only two values adequate and inadequate for fund status which is declared as a free type. The constraints for the customer status

sig fund_status {}

sig adequate, inadequate extends fund_status {}

fact constraint_customer { **all** t : Account | t.status = = adequate ⇔

{ (t.atm_funds & t.atm_critical_value) & (t.loan_funds & t.loan_critical_value) ||

(t.atm_funds & t.atm_critical_value) & (t.bal & t.acc_critical_value)}}.

2. **The maximum number of trials for invalid thumb extension is 3.**

module Thumb_Extension_Module/invar

open util/ordering[ThumbreenterState] as so

open util/integer as integer

sig ThumbreenterState { count: Int, // integer variable }

fun inc [n : Int]: Int { add [n,Int[1]] }

pred init { **let** fs = so/first | { fs.count = Int[0] }}

pred extend [pre, post: ThumbreenterState]

{ **some** X: Int | pre.count = X and post.count = inc[X] }

fact createThumbreenterStates { init all s: ThumbreenterState - so/last | let s' =

so/next[s] | extend[s,s'] }

run {} for **exactly** 3 ThumbreenterState, 3 **int**.

Figure 5.5, shows the different state change for the thumb authentication. In first trail the count is 0 and if authentication is not succeeded then it extend to re enter thumb state. When count becomes 3 it does not allow the customer to retrying the process.

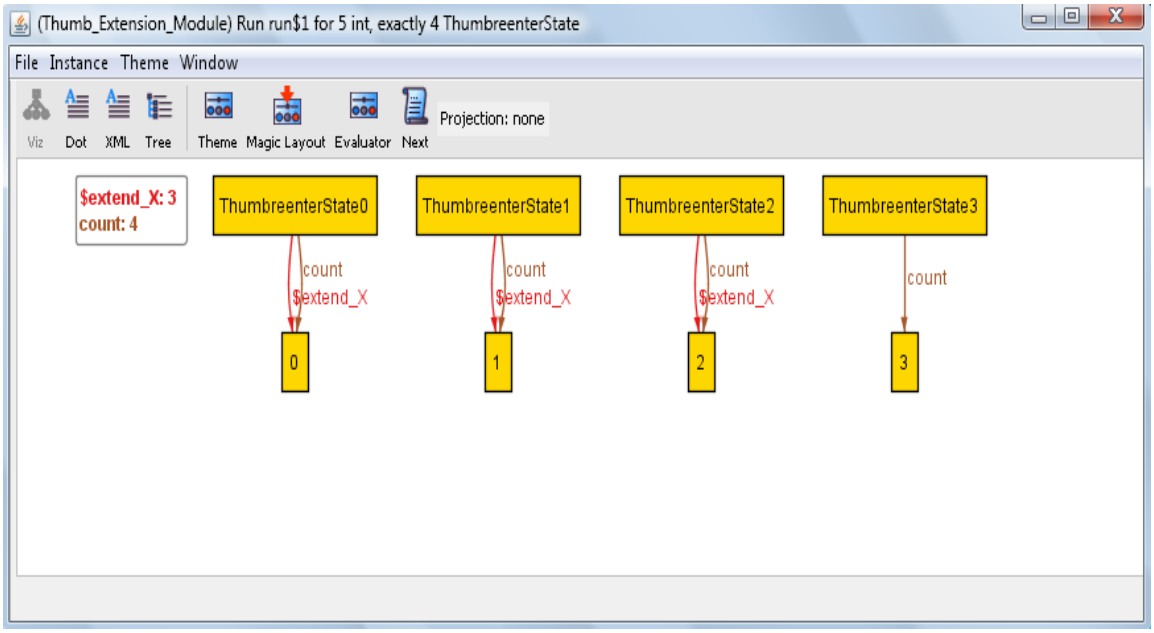

Figure 5.5: Visualization view for invalid thumb extension is 3.

3. **Every card presented in the data base should be mapped with at most one thumb.**

module Unique Thumb

sig thumbimpression, card{ }

sig thumbdatabase { has1:thumbimpression one ->one card }

pred show [b: thumbdatabase] { #b.has1 > 1}

run show for 6 **but** 1 thumbdatabase.

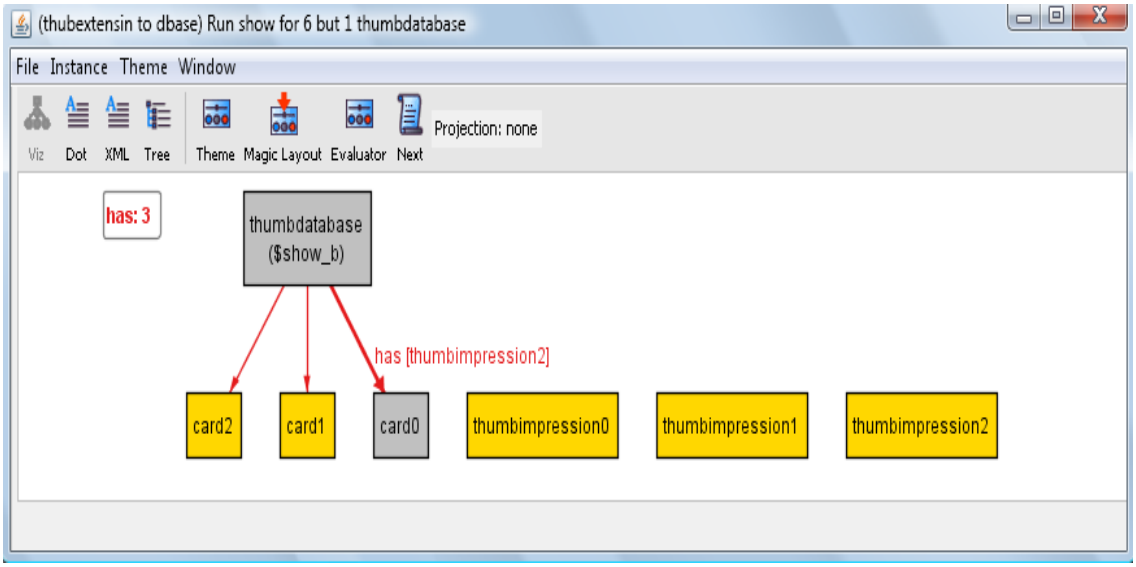

Figure 5.6: Uniqueness representation for Card-Thumb mapping.

4. **When authentication succeeded the menu of option is displayed. If customer selected one option then it should not match with other.**

Module menu

Open util/ordering [menu_state] as org

One sig menu {}

Sig menu_state { done, balance, withdraw, deposit, exit : set menu}

Fact exclusive {no m_1, m_2 : menu | m_1 = menu && m_2 = menu && disj [m_1, m_2]}.

// Initially the transaction log file did not contains the any option.

Fact initial_menu_state { so = ord | **first** | no so.done && (so.balance =

menu && so.deposit = menu && so.withdraw

= menu && so.exit = menu)}.

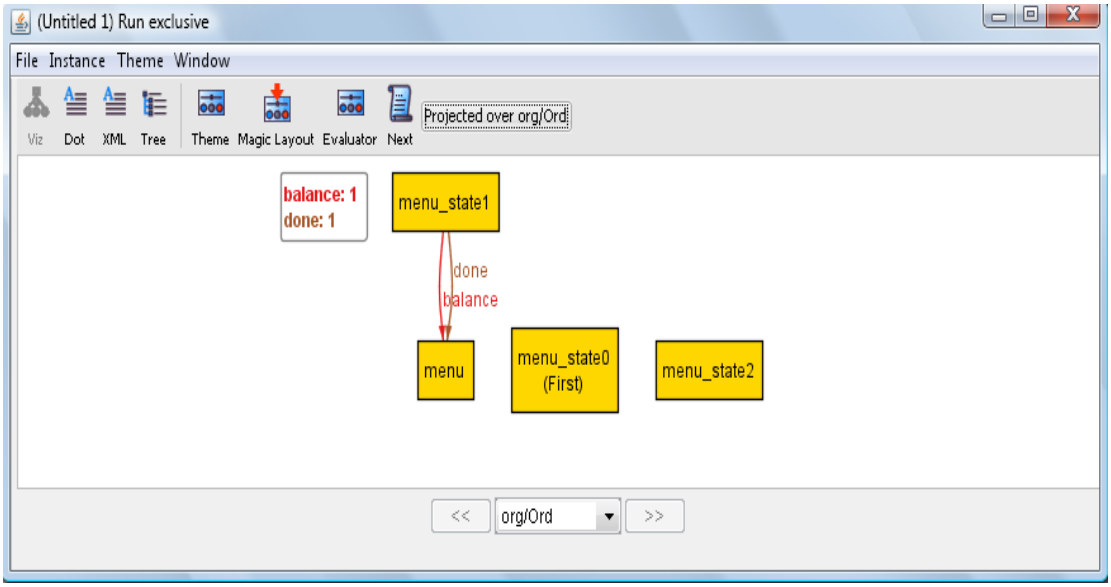

Figure 5.7: One menu option did not match with other.

After the authentication the ATM displays the menu of options, from that options the customer has choice to select any option or cancel the selected one or cancel the selected one and selects a new choice from the menu. When the customer chooses an option from the menu ATM adds that choice to menu cart for further process. If the customer makes a choice of cancelling it then ATM removes that option from the process. If the customer reselects any option then it replaces the selected option in the cart. So, every time ATM performs at most one operation among (add, move or remove) them from the cart. Adding an option

sig State { contains: menu -> Option}

fun add_menu_option (**disj** s, s' : State, disj op_1,o_1 : Option)

{(op_1->o1!in s.contains) && (s'.contains = s.contains + op_1->o_1)}.

Here, o_1 represents the option to be added to a menu op_1 in the after state s'. This relationship did not exist in the before state which is expressed as op_1->o_1 !in s.contains. The frame condition s'.contains = s.contains + op_1->o1 ensures that every other relation remains the same and that the option is added. The keyword disj indicates that any two options are disjoint. It can be argued that the precondition op_1->o_1 !in s.contains is not needed and whether the && (and) symbol should be replaced by an implication. Note further, that s and s' have no defined meaning in Alloy unlike, for example, s' would have in a notation like Z [10].

run add_menu_option for 4 **but** 2 State.

// Removing an option.

fun remove_menu_option (disj s, s' : State, disj op_1,o_1 : Option)

{op_1->o_1 in s.contains && (s'.contains = s.contains - op_1->o_1)}.

// as a third operation the moving of a selected operation between two menu options
// can be specified as follows

fun move_menu_option (disj s, s' : State, disj o_1,op_1,op_2 : Option)

{(op_1->o_1 in s.contains && op_2->o_1 !in s.contains) &&

(s'.contains = s.contains - op_1->o_1 + op_2->o_1)}.

However, the above operation may not be required since the elementary operations of removing and adding objects can be composed to model the moving of an object between management menus. A sequence of menu operations expressed as an explicit signature

sig State_Sequence {**disj** first, last : State, next: (State - last) !->! (State - first)}

{//Constraints on the signature

all s : State | s in first.*next

all states : State - last | **some** s = states | some s' = states.next |

(**some** op : Option | some d1,d2 : menu |

add_menu_option (s, s', op, d1) (//add an object) || //or

remove_menu_option (s, s', op, d1) (//remove an object) || //or

move_menu_option (s, s', obj, d1, d2) (//move an object)) }.

5. **Add thumb and thumb authentication should be inherited from the thumb data base.**

   module add_thumb

   sig thumbimpression, card{ }

   sig thumbdatabase {has:thumbimpression one ->one card}

   pred add [b, b': thumbdatabase, n: thumbimpression, a: card]

{ b'.has = b.has + n->a }

pred showAdd[b, b': thumbdatabase, n: thumbimpression, a: card]

{add [b, b', n, a]

#thumbimpression.(b'.has) > 1}

run showAdd for 3 **but** 2 thumbdatabase.

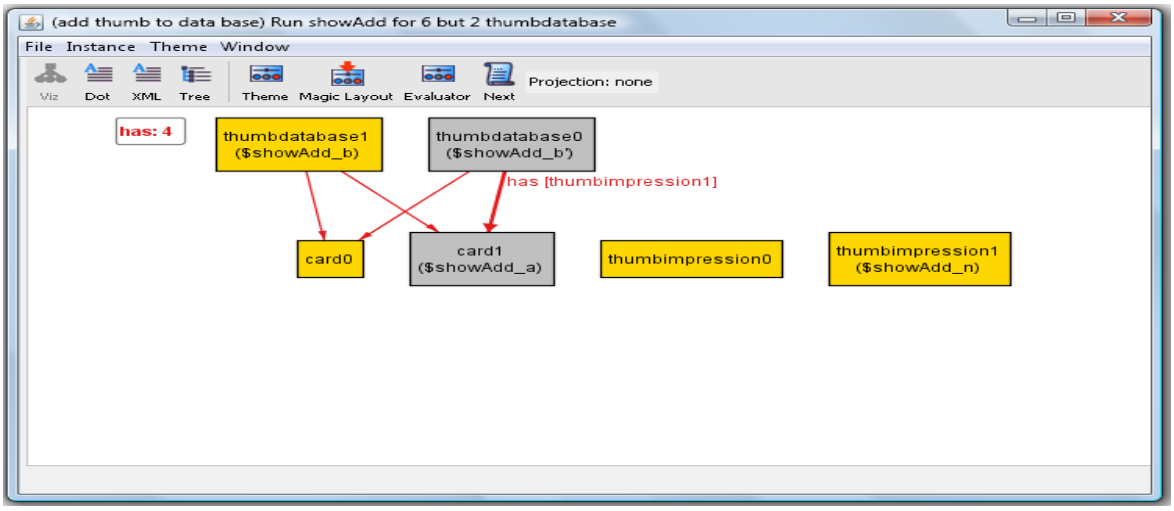

Figure 5.8: Map relation between card and thumb in the data base1.

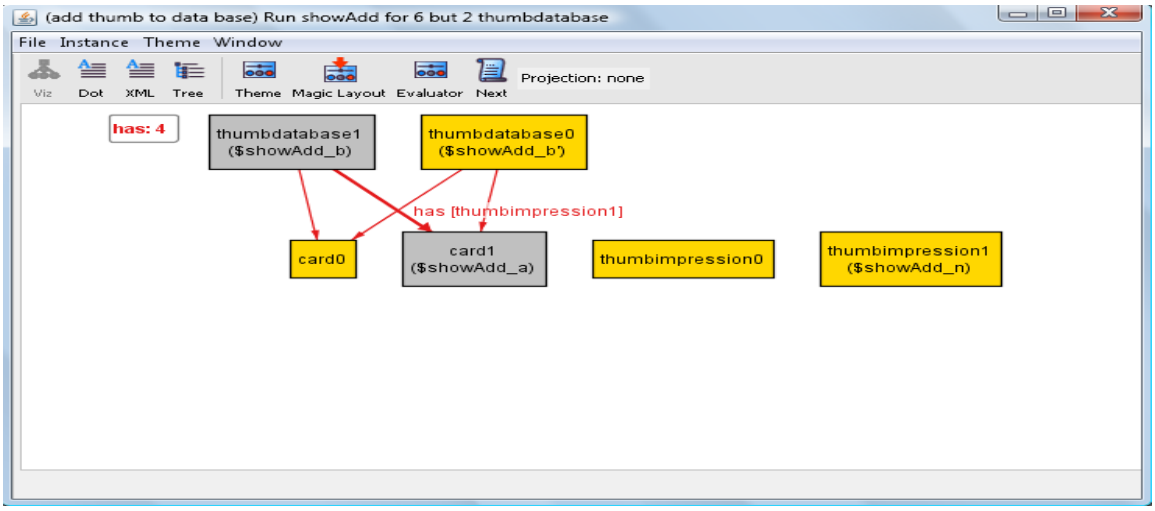

Figure 5.9: Map relation between card and thumb in the data base2.

Suppose there are two thumb data bases. So adding of thumb data can be done in any data base. From the above figures (5.8, 5.9) it shows that there two cards and two thumb impressions. In the both data bases that $card_1$ has a relation with only one thumb $impression_1$.

Thus, all specifications of ATM system are modelled here successfully using Alloy.

Chapter 6
Combining Z specification to C++ code

This approach enables to analyze the combination of specification and implementation. By combining code and specification, main intention is to provide a prototyping facility by integrating the execution of code into the interpretation of a specification. The mapping transforms schemas of Z specification into classes (more precisely, class interfaces) of C++ such as data members and headers of member functions. Therefore, distinguish between the specification level, which refers to the Toolbox, and the code level which relates to the integration of code for the execution. That is, it is not aimed as a code generation system, but kind of a tool for analyzing specification (including syntax, semantic and type checking) and for helping a software developer in obtaining code from specification.

The combining of C++ code and specification in the Z language and its basic idea are shown in figure 6.1. In figure 6.1, there are two different levels. One is dashed boxes and second is bold face boxes. The dashed boxes represent the different levels, the Z specification tool box represents the specification level and the code level corresponds to the code units. These code units are represented as “dynamic linked libraries" to the interpreter process.

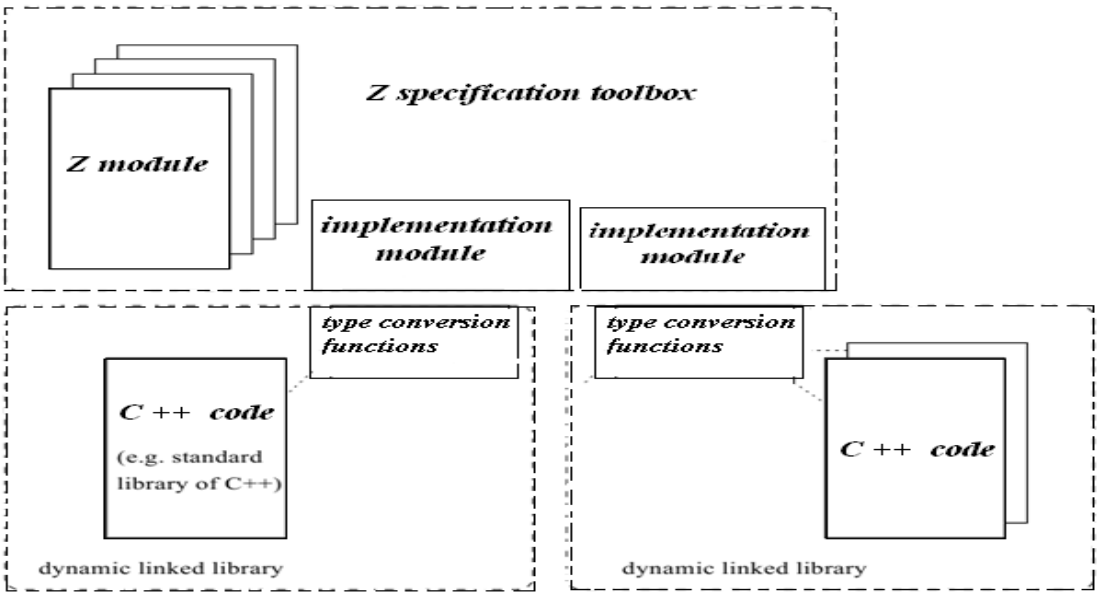

Figure 6.1: Basic idea of combining code and specification in the Toolbox[28].

The bold face boxes show the parts which the user has to develop additionally in order to combine code and Z specification. For every unit of code, which may consist of one or more C++ files, an interface at the specification level as well as an interface at the code level must be developed by the user of the toolbox. At the specification level a new kind of module, called implementation module, needs to be introduced. The module concept of the specification language provides a facility for the combination of multiple modules, which has been used for this approach.

The type information corresponding for every definition of the C++ code to be integrated with the Z specification is held in an implementation module at the code level. The implementation modules have to be imported by every module which access a function or value defined in code.

As there are two different levels having different representations for the Z specifications and C++ code for their types, a function should be established between them. They are known as type conversion functions. This function takes as argument values of the translator process and converts these to values required by integrated code.

6.1 Basic Class

In order to implement the idea, need to define the structure of Z module which is used as a class to describe specification models. The structure of the class consists of several parts (Figure 6.2):

Class name: is the name of the main schema which should be structurally map, includes precondition, post condition and invariant functions.

Type definitions: schema variables declaration.

Functions: These are the Δ notation in operation schemas (which indicates the change of values of states by the operation); i. e includes the schema names which change the state of the main (class) schema.

State variables: These are the schemas which contains only the declaration of the Variables and Predicates.

Global variables: It is a schema without the schema name which contains the declaration of variables and some constant values, and available for all schemas.

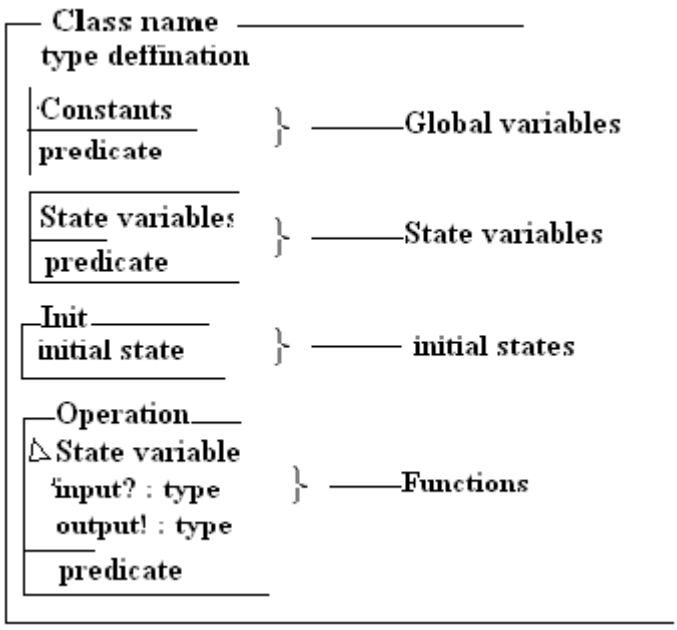

Figure 6.2: Z module specification [23].

6.2 C ++

The programming language C++ is an extension of C, and many language features mainly related to object-oriented paradigm are added [32]. There are three parts in the members of a class of C++ concerning accessibility to these from outside: public, protected and private. And also, one can control the accessibility in a derived class when inheriting the members of a base class, using the same keywords public, protected and private. And it also has a friend feature.

6.3 Structural Mapping

6.3.1 Basic rules

1. Global variables, state variables and constants in a Z module class are mapped into the protected part of a C++ class.
2. All schemas which change the state (inheritances) in Z module are mapped to public inheritances of C++.
3. Functions in Z module classes are mapped to virtual functions in C++. The return values of the functions are of type void, and their parameters are passed by reference.

4. For each class of C++, a null constructor, a copy constructor, a destructor, an assignment operator, and invariants for constants are always supplied.
5. All member functions (corresponding to Z operations) are declared as virtual functions in C++.
6. Constructors for types of constants are always supplied.
7. In the Z module (fig 6.2) the main class considered as base class and the schemas which are changing the state of the main schema are considered as derived classes (functions). Similarly identify the multiple inheritances.
8. Z operators related to power sets and partial functions are prepared as Classes Power and PFun (templates) in C++ respectively.
In basic rule 1 the intention is that data members of a C++ class be encapsulated from outside, but in some cases it would be more preferable to map constant and state variables of an Z module class to the public part of a C++ class.

6.3.2 Type Conversion

To assist with conversion of types, develop a class library that implements some of the standard types of the Z mathematical tool kit. This library provides implementation of the set, sequence, relation and function type. The implementation is in the form of generic classes. But generosity itself can be realized in C++ by the template construct. The generic symbol of power set and partial functions are realized as a predefined class in C++, as **template** <class name> object and **template** <set1, set2>function name respectively.

For example, the definition **power** <NODE> Nodes; is a generic class corresponding to the Z definition Nodes: ℙ NODE. Thus, solution for realizing these type construction methods in C++ is simple; map each of these type constructions directly to a (template) class of C++. There are many ramifications of actually realizing these classes in C++, which differ to each other in simplicity and efficiency [33].

6.4 Architecture of Z-C++

The Architecture of Z-C++ is illustrated in Figure 6.3 .The figure illustrates how the Z toolbox and C++ are connected as well as the flow of data between the tools. Starting at the lower left corner of the figure 6.3, it shows how a set of Z files (in

Latex format) are parsed and added to the class repository of the Z toolbox. The link translates the Z specification into an intermediate representation capturing the parts of Z module relevant in C ++. In the same way, by accessing the class repository of C++, the class model is translated to the intermediate representation. The two intermediate representations are "compatible" and can thus be compared or merged. Merging the two models into one common model, and subsequently propagate this model to the class repository in C++ and the Z specification files, will synchronizes the Z and C++ model. If there is no intermediate representation resulting from C++ or if one chooses to ignore such a representation, the merger will result in a full translation from Z to C++. Parallel, if there is no intermediate representation resulting from the Z toolbox or if one chooses to ignore such a representation, the result is a full translation from C++ to Z.

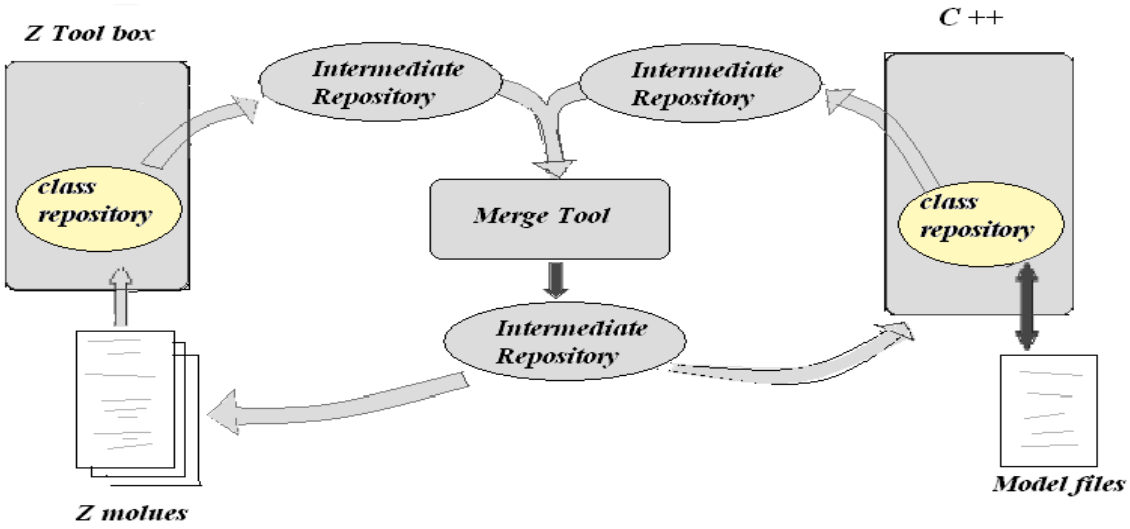

Figure 6.3: The architecture of the structural mapping of Z-C++ [29].

6.4.1 An example of mapping

As an illustration of structural mapping, here gives an example of a Z specification and its corresponding C++ code of declarations generated by mapping system. Figure 6.4 shows the specification of a Bank system. The main functions are Balance enquiry, Cash Withdrawal, Deposit Money. A schema of Bank is introduced which contains a state schema and three operation schemas, is mapped to a C++ class, consisting of class interfaces, i.e. declarations of data members and headers of member functions of the class. Note that the suffixes "-i" and "-o" of the

parameters of the operations Deposit and the others. They correspond to the decorations "?" and "!" of variables in operation schemas of Z module.

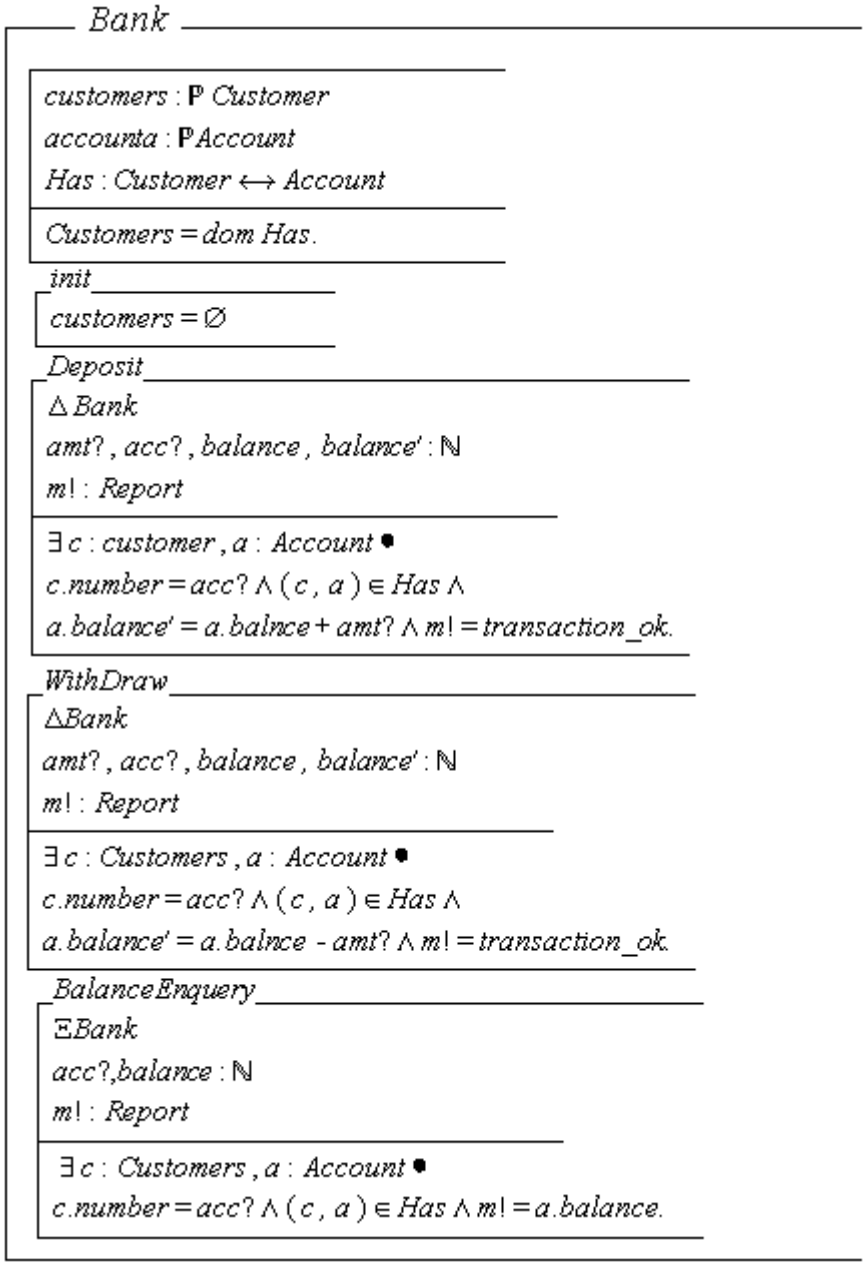

Figure 6.4: Bank module.

The result of the mapping specification into code:

```
#include "GlobalDefs.h"
class Bank {
protected:  //Declaration Part
Power< Customer > customers;
PFun< Customer, Account > Has;
Public: Bank (); // Null constructor.
Bank (Bank&  the-Bank); //copy constructor.
```

```
Virtual ~Bank (); // Destructor.
Bank& operator = ( Bank& the_Bank); // Assignment Operator.
Virtual void Deposit (Account& acc-i, Amount& amt-i, Report& r-o);
Virtual void Balance Enquiry (Account& acc-i, Report& r-o);
Virtual void Withdraw (Account& acc-i, Amount& amt-i, Report& r-o);
```

6.5 Implementation of a Mapping System

6.5.1 LaTeX mark-up

The mathematical representation of Z is what one would write with pen, pencil, chalk, etc. Instructing a computer to produce the same appearance currently requires the use of a mark-up language. There are many different mark-up languages, tailored to different circumstances, such as particular typesetting software. Specifications of Z for a mapping system must be in a machine readable format.

LaTeX is a document mark up language and document preparation system for the TeX typesetting program. The term LaTeX refers only to the language in which documents are written, not to the editor used to write those documents. In order to create a document in LaTeX, a .tex file must be created using some form of text editor. While most text editors can be used to create a LaTeX document, a number of editors have been created specifically for working with LaTeX. All LaTex mark up needs to be terminated by a \ character: this is to try and ensure that it the characters are entered in the correct font and style. The LaTex source code for the bank and the deposit schema is:

```
\begin{schema}{Bank}
Customers~:~\power~Customer\\
Accounts~:~\power~Account\\
Has~:~Customer~\rel~Account\\ \where
Customers~=~\dom~Has\\ \end{schema}
\begin{schema}{Deposit}
\Delta Bank\\
amt?~,~acc?~,~balance~,~balance'~:~\nat\\
m!~:~Report\\ \where
\exists~~c:~customer~,~a~:~Account~@\\
```

c.number~=~acc?~\land~(~c~,~a~)~\in~Has~\land\\
a.balance'~=~a.balance~+~amt?~\land~m!~=~transaction_ok.\\ \end {schema}.

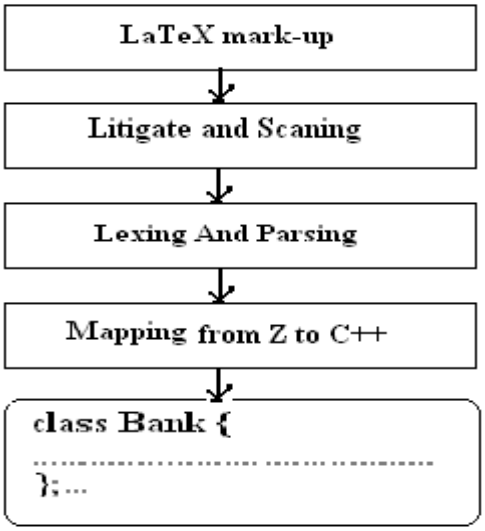

Figure 6.5: Mapping.

6.5.2 Litigate and Scanning

As litigating for mapping, some parts in input specifications are removed, which are the following.

1. LaTeX commands such as \documentstyle {} and \begin (document).
2. Informal verbal explanations other than Z specification. In LaTeX form they are the parts that are not within environments such as \begin {class} .. .\end {class} and \begin {zed} ... \end {zed}.
3. Comments within specifications. They are in \comment {. . .}, \comment *{. .}, and \begin {z par} ... \end {z par}.
4. Newline symbols (\\) that specifies infix operators.

The structure of the name table is as in Fig 6.6. All the names appearing in an input specification in Z specification are recorded in the name table. First identify the global variables. These global names (particularly class names) are maintained in a linear list in the order they have appear in the source specification. Purpose is to check the scope of the names. The global names can be accessed in both ways by linear list and by hashing. Local names in z specification and schemas are considered as attributes of a global name of a class or a schema they belong to, and are linked in a list starting from a global name. There are several kinds of entries (class names, state names, operation names, . . .) in the name table [figure 6.6, 6.7].

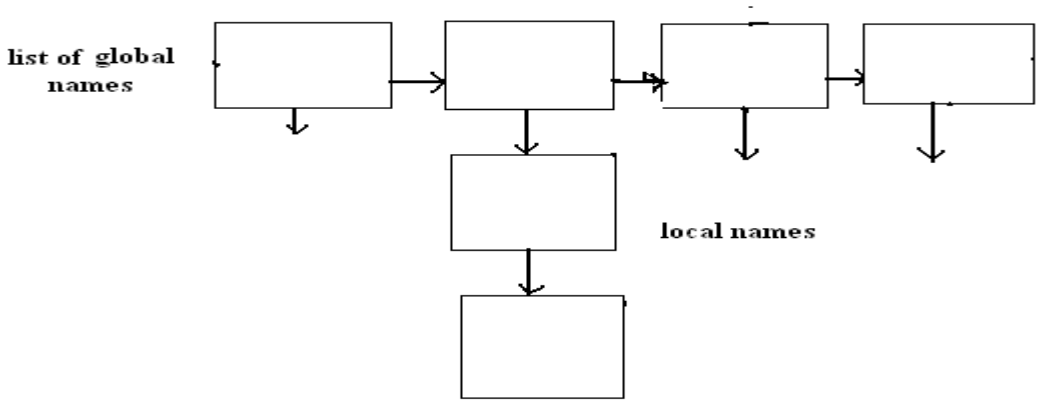

Figure 6.6: structure of name table.

Char* Name
:
List base classes
List type def and enums
Constants
List of states
List of operation schema

Figure 6.7: Structure of Class Attributes.

6.5.3 Lexing and Parsing

Lexing is the translation of a specification's sequence of Z characters to a corresponding sequence of tokens. The schema (SCH) definition paragraph SCH i t END introduces the global name i, associating it with the schema that is the value of t.

SCH i t END ⇒ AX [i = = t] END

The paragraph is semantically equivalent to the axiomatic (AX) description paragraph whose sole declaration associates the schema's name with the expression resulting from syntactic transformation of the schema text. Where t denotes a Schema Text phrase (t for text) and i denotes identifier of schema. Associated with the tokens NAME and NUMERAL are the original names and numerals. In figure 6.8 on the left is the sequence of tokens corresponding to the extract from the birthday book, and on the right is the same sequence but revealing the underlying spelling of the name tokens.

[NAME, NAME] END	[NAME , DATE] END
SCH NAME	SCH BirthdayBook
NAME : P NAME NL	known : P NAME NL
NAME : NAME I NAME	birthday : NAME ⇸ DATE
NAME = NAME NAME	known = dom birthday
END	END

Figure 6.8:Translation of a specification's [16].

Parsing is the process of analyzing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given (more or less) formal grammar. In computing, a **parser** is one of the components in an interpreter or compiler, which checks for correct syntax and builds a data structure (often some kind of parse tree, abstract syntax tree or other hierarchical structure) implicit in the input tokens. Parsers may be programmed by hand or may be (semi-)automatically generated (in some programming languages) by a tool (such as Yacc or Lex) from a grammar written in Backus-Naur form (BNF). A BNF specification is a set of derivation rules, written as <symbol> ::= expression.

Where <symbol> is a nonterminal, and the expression consists of one or more sequences of symbols; more sequences are separated by the vertical bar, '|', indicating a choice, the whole being a possible substitution for the symbol on the left. Symbols that never appear on a left side are terminals. On the other hand, symbols that appear on a left side are non terminals and are always enclosed between the pair <>.

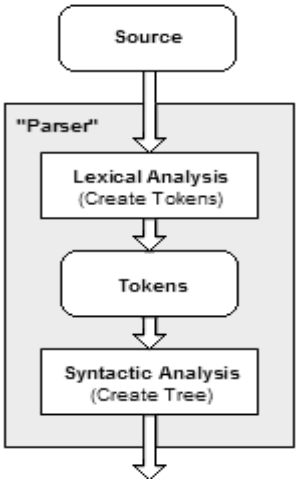

Figure 6.9: Overview of parser process [21].

The first stage (figure 6.9) is the token generation, or lexical analysis, by which the input character stream is split into meaningful symbols defined by a grammar of regular expressions. For example, a calculator program would look at an input such as "`12*(3+4)^2`" and split it into the tokens `12`, `*`, `(`, `3`, `+`, `4`, `)`, `^`, and `2`, each of which is a meaningful symbol in the context of an arithmetic expression. The lexer would contain rules to tell it that the characters `*`, `+`, `^`, `(`and`)` mark the start of a new token, so meaningless tokens like "`12*`" or "`(3`" will not be generated. The next stage is parsing or syntactic analysis, which is checking that the tokens form an allowable expression. The syntax rules of Z are taken from the [10].

6.5.4 Mapping

The meaning of a Z specification is established by relating it to a reading in a semantic world. Syntax for the mapping class as follows

Specification = {Section}, NAME , Parents

| SCH,AX , Schema Text , Expression, END

| {Paragraph}.. .....;

An example in mathematics, a rather liberal notation is used for writing down polynomials in x, such as $x^3 - 2x^2$. The following grammar describes such polynomials:

```
Poly      = Term
          | Plusminus Term
          | Poly Plusminus Term .
Term      = Natnum "x" Exponent
          | Natnum
          | "x" Exponent .
Exponent = "^" Natnum
          | Λ .
Plusminus = "+" | "-" .
```

A prototype version of a mapping system from specification to code is a useful tool, when checking and analyzing specification and rewriting specifications to code.

Chapter 7
Conclusion and Future work

7.1 Conclusion

Following conclusions are drawn all the basis of work done in this thesis:

- Z is one of the numbers of specification languages which are being developed around the world. Z can be used to compactly specify real systems (ATM). Z has various collection of library (Mathematical Toolkit), which supports user to specify the requirements without any ambiguity.
- Large specifications are achievable in Z, using the schema notation for structuring. Also it is possible to produce hierarchical specifications. A part of a system is specified in isolation, and then put into a global context.
- By applying formal method in terms of Z notation, it is observed that it does not require a high level of mathematics rather it requires knowledge of basic set theory and first order logic for the analysis of a complete system.
- Difficulties with Z are cannot do concurrency, Timing aspects, Algorithmic aspects and programming constraints, and Sequencing operations. Definition of duration calculus concepts as they apply to specification in Z is given. In this thesis Alloy model is used for handling inefficiency of Z language. Alloy is a first-order logic modelling language with tool support for fully automated syntactic and semantic analysis, which generates an example of an under constrained model.
- It is possible that the schemas of Z specification can be transformed into classes (more precisely, class interfaces) of C++.

7.2 Future Work

- Investigating formal refinement techniques for Real-Time Systems.
- Some of various features of Z are not implemented to C++ library. These are relational symbols, predefined functions and sequence (like concatenation) should be mapped into the C++ class library.

References

[1] Zhou Chaochen, C.A.R.Hoare, and A.P.Ravn, "A Calculus of Durations", In Information Processing Letters 40(5), 1992, pp. 269-272.

[2] R. Duke and G. Smith, "Temporal logic and Z specifications", The Australian Computer Journal, 21(2):62-66, 1989.

[3] I. Houston and M.Josephs, "Specifying distributed CICS in Z", accessing local and remote resources, Formal Aspects of Computing, 6(6), 1994.

[4] C. W. Johnson, "Using Z to support the design of interactive safety-critical systems", Software Engineering Journal, March 1995.

[5] Maritta Heisel and Jeanine Souquieres, "A Method for Requirements Elicitation and Formal Specification", Springer-Verlag, ISBN:3-540-66686-9, Vol. 1728, pp. 309 - 324, 1999.

[6] Martin Giese and Fogardt Heldal, "From informal to Formal Specification in UML", Proceedings of UML 2004, LNCS 3273, pp. 197--211, 2004.

[7] Munina Yusufu and Gulina Yusufu, "Comparison of Software Specification Methods using a Case Study", International Conference on Computer Science and Software Engineering, 2008.

[8] Information technology Z formal specification notation Syntax, type system and semantics, ISO/IEC 13568:2002(E), International Standard.

[9] Jim Woodcock and Jim Davies "Using Z Specification, Refinement, and Proof", Formal Aspects of Computing Journal 7(3):266–288 1995.

[10] J.M. Spivey, "The Z Notation, A Reference Manual", 2nd edition, Prentice Hall International, 1992.

[11] Wang Y. And Y Zhang, "Formal Description of an ATM system by RTPA", Proceedings of CCECE'03, IEEE CS Press, pp. 1255-1258, 2003.

[12] J.M.Wing, "A Specifier's Introduction to formal methods", IEEE Computer, vol. 23, no. 9, September 1990, *pp. 8-24*.

[13] P. Baumann and K. Lermer, "A Framework for the Specification of Reactive and Concurrent Systems in Z", Proceedings of the 15th Conference on the Foundations of Software Technology and Theoretical Computer Science, LNCS 1026,62-79, Springer Verlag, 1995.

[14] M. Lindahl, P. Pettersson, and Y. Wang, "Formal Design and Analysis of a Gearbox Controller," Springer Int'l J. Software Tools for Technology Transfer, vol. 3, no. 3, pp. 353-368, 2001.

[15] Sabnam Sengupta and Swapan Bhattacharya, "Formalization of UML Use Case Diagram-A Z Notation Based Approach", IEEE, ISEC'08, February 19-22, 2008.

[16] "Formal Specification-Z Notation Syntax, Type and Semantics Consensus", Developed by members of the Z Standards Panel BSI Panel IST/5/-/19/2 (Z Notation) ISO Panel JTC1/SC22/WG19, Project number JTC1.22.45, August 24, 2000.

[17] Chris Matthews, "Fuzzy concepts and formal methods: A Sample Specification for a Fuzzy Expert systems", Proceedings of the World Congress on Computational Intelligence (WCCI 2002), IEEE Press, 2002.

[18] Peter Baumann and Karl Lermer, "Specifying Parallel and Distributed Real-Time Systems in Z", Proceedings of the 4th WPDRTS, IEEE 1996.

[19] Hyun-Jeong Jo, Jong-Gyu Hwang, and Yong-Ki Yoon, " Formal Requirements Specification in Safety-critical Railway Signaling System", Transmission Distribution Conference Exposition: *Asia* and Pacific, 2009.

[20] Manpreet Singh and Manjeet .S. Patterh, "Access Control Framework for Secure Network Computing Environment", IEEE International Computer Software and Applications Conference 0730-3157/08, 2008 .

[21] http://en.wikipedia.org/wiki/Parsing , Mar 2010.

[22] Bowen J.P, "Formal Specification and Documentation of Microprocessor Instruction Sets", Microprocessing & Microprogramming archive. Volume 21, Issue 1-5, August 1987.

[23] Mohammad Mousavi, "Software Specification Functionality Specification in Z", Design and Analysis of Systems (OAS) Group Department of Computer Science TU/Eindhoven September 15, 2009.

[24] C. C. Morgan and J. W. Sanders, "Laws of the logical calculi", Software Engineering Journal, 4(2):74{*78)*, March. 1989.

[25] JACKSON D, "Software Abstractions – Logic, Language and Analysis", Cambridge: The MIT Press, 2006.

[26] M. Joseph, "Formal Techniques in Real-Time and Fault-Tolerant Systems", Lecture Notes in Computer Science 331, pp 160-174.

[27] Khurshid, S., Marinov D, and Jackson D, "An Analyzable Annotation Language", ACM SIGPLAN Notices, New York, v. 37, n. 11, p. 231-245, 2002.

[28] Brigitte Frohlich and Peter Gorm Larsen, "Combining VDM-SL Specifications with C++ code", FME 1996: pp 179-194.

[29] http://www.vdmtools.jp/en/modules/news/index.php, 15 sept 2009.

[30] URL http://vl.fmnet.info/ .

[31] Roger S.Pressman, "*Software Engineering*- A Practitioner's Approach", McGraw Hill, 5th edition. 2000.

[32] M. A. Ellis and B. Stroustrup, "The Annotated C++ Reference Manual", Addison-Wesley, ISBN 0-201-54330-3, 1990.

[33] H. Miyazaki, k. Yatsu, M. Soineya, S. Yamasaki , and K. Kakelii, "The library problem", in Deductive Derivation of programs, preliminary report, pp. 239-362, Information Promotion Agency, Japan, 1992.

[34] URL http://spd-web.terma.com/Projects/RAISE/ .

[35] The Software Design Group at M.I.T, URL to The Alloy Analyzer webpage, http://alloy.mit/edu/ Feb 2010.

List of Publications

Published:

Sathish Kumar M and Shivani Goel, “Specifying Safety critical Real-Time Systems in Z”, International Conference on Computer and Communication (ICCCT), IEEE 2010.