Sudan University of Science and Technology
College of Graduate Studies

Automated Verification of Abstract Data Types

التحقق الآلي من أنواع البيانات المجردة

A dissertation Submitted in Partial Fulfilment of the Requirements for the Degree of Doctor of Philosophy

In

(Computer Science)

BY

NAHID AHMED ALI AHMED

Supervisor

Professor: Ali Mili

JUNE 2017

DECLARATION

Some of the material in this dissertation have been previously published in journal and conference papers; the details are given below:

1. Nahid Ahmed Ali, Verifying Abstract Data Types: A Hybrid Approach; 1st International Conference On Computing, Electrical And Electronic Engineering (ICCEEE 2013), Page(s): 634 – 639, 2013.
2. Nahid A. Ali, التحقق من انواع البيانات المجردة: مدخل هجين, International Arab Conference on Information Technology (ACIT' 2015), 2015.
3. Nahid A. Ali, Amal A. Mirghani and Abdelrasoul Y. Ibrahim, Alneelain: A Formal Specification Language, 2017 International Conference on Communication, Control, Computing, and Electronics Engineering (ICCCCEE), Page(s): 1 – 9, 2017.
4. Nahid A. Ali, A Survey of Verification Tools Based on Hoare Logic, International Journal of Software Engineering & Applications (IJSEA), Vol.9, No.2, 2017.
5. Nahid A. Ali, Verifying ADT Implementations against Axiomatic Specifications, International Journal of Mathematics and Statistics Invention (IJMSI), Vol.5, No.3, 2017.

I declare that no part of this material has previously been submitted to a degree or any other qualification at Sudan University of Science & Technology or other institutions. I further declare that this thesis is my original work, except for clearly indicated sections where appropriate attributions and citations are given to work by other authors.

Nahid Ahmed Ali Ahmed

Signature: Nahid Date: 15/06/17

Sudan University of Science and Technology

College of Graduate Studies

DECLARATION

I, the signing here-under, declare that I'm the sole author of the (M.Sc./Ph.D.) thesis entitled:.. ...., which is an original intellectual work. Willingly, I assign the copy-right of this work to the College of Graduate Studies (CGS), Sudan University of Science and Technology (SUST). Accordingly, SUST has all the rights to publish this work for scientific purposes.

Candidate's name:

Candidate's signature:

Date

DEDICATION

To the soul of my Father:

The first to teach me, who would be delighted if he saw me in what I am in now

To my beloved Mother:

For her prayers for me and for her constant, unconditional love and support.

To my Brothers and my lovely Sister

To the people who add meaning to my life

To everyone who has supported me during this research

ACKNOWLEDGEMENT

In the Name of Allah, the Most Gracious, the Most Merciful.

First and foremost, I thank Almighty God for everything in my life and for giving me the courage and the determination, as well as guidance in conducting this research, despite all the difficulties.

I am grateful to my supervisor, Professor Ali Mili, for his invaluable support, guidance, and advice throughout my PhD. His incredible attention to detail, technical understanding, and unending enthusiasm have been of great benefit to me – and this dissertation– several times over. Our research discussions were always very lively and enjoyable.

I owe thanks to the General Directorate for External Relations and Training - Ministry of Higher Education and Scientific Research for their financial support throughout my doctoral study.

Dr. Abuobeidah Hamouda deserves special thanks for carefully reading through early drafts of this dissertation.

I also express my gratitude to Mr. Mojtaba for his technical support and programming in this research.

I thank my family for their unwavering love, encouragement, and support throughout these four years as a student. I am really incredibly lucky to have you all.

Warm thanks go to my dearest brother, Nadir and his lovely family; I will never be able to repay him for his absolute backing and support, meeting my needs before I knew I needed them. He has a special place in my heart.

I wish to mention all my other brothers and my sister, who have all had a great influence in keeping me motivated throughout this PhD journey. When things made no sense, they made the PhD life looks all the brighter and gave me the extra push when I needed one.

I am particularly grateful to my fiancé Hassan and his family who have kept exemplary patience while I completed my thesis. I am indeed blessed to have them in my life.

Lastly, I offer my regards and blessing to all those who supported me in any respect during the completion of this work as well as expressing my apologies that I could not mention them personally one by one.

ABSTRACT

Despite the fact that modern computer systems are composed of complex hardware and software components, verifying the correctness of the software part is often a greater problem than that of the hardware. It is known that manual inspection of complex software is error-prone and expensive, therefore tool support is required.

According to practical experience, using Hoare logic for proving the correctness of computer programs is tedious, error-prone, obscure and entirely unreliable process. This is because correctness-based principles are generally not well understood. No doubt that this critical state of affairs has numerous reasons. However, there is one reason or issue that has greatly led to this problem that is the verification tools used in applying Hoare logic. These tools, in many cases, consist of a pen and papers, making it a tedious task to verify a whole program using a pen and a sheet of paper. A direct way to solve this problem is to use automated formal verification software to facilitate checking the correctness of Hoare programs. The aim of this research is to build an automated system called *Alneelain* Verification System that maps axioms of Abstract Data Types (ADT's) specification into verification conditions (in the form of a Hoare formula) and attempts to prove them in Hoare's logic. Before mapping axioms into Hoare formulas, one needs to check that the specification is syntactically correct. In order to do this a specification language called *Alneelain* is developed based on axiomatic specification. The evaluation results show that the developed verification system provides a high degree of proof automation combined with the ability to provide feedback on failed proof attempts and thus removes the difficulty associated with the process of applying Hoare logic manually.

مستخلص البحث

بالرغم من حقيقة أن نظم الحاسوب الحديثة تتكون من عتاد وبرمجيات معقدة ألا أن التحقق من جزء البرمجيات يكون غالباً مشكلة اكبر من تلك التي تخص جزء العتاد. ومن المعلوم أن التحليل اليدوي للبرمجيات المعقدة مكلف وليس خاليا من الأخطاء. لذا فإن دعم الأدوات يكون مطلوباً. بناءاً على التجارب العملية، فإن استخدام منطق هوار لإثبات صحة برامج الحاسوب يعتبر عملية مضجرة وقابلة للخطأ وقاتمة وغير موثوق بها بالمرة. وذلك لأن مباديء التحليل المنطقي للبرمجيات عموماً ليست مفهومة بصورة جيدة. من دون شك فإن هذه الحالة الحرجة لها العديد من الأسباب ولكن هنالك سبب واحد أدى إلى هذة المشكلة وهو أدوات التحقق التي تستخدم في تطبيق منطق هوار. هذه الأدوات في حالات عديدة تتكون من قلم و اوراق مما يجعل مهمة التحقق من برنامج كامل باستخدام القلم والورق مملة. الطريقة المباشرة لحل هذه المشكلة هي أستخدام برمجيات التحقق الممنهج الآلية للمساعدة في التحقق من صحة برامج هوار. هدف هذه الدراسة هو بناء نظام آلي سمى نظام النيلين للتحقق يحول البديهيات الخاصة بمواصفات أنواع البيانات المجردة الى شروط التحقق (في شكل صيغة هوار) ومن ثم يحاول إثباتها في منطق هوار. قبل وضع البديهيات في صيغة هوار، يحتاج الفرد إلى التأكد من أن المواصفات صحيحة من ناحية التركيب. ومن أجل فعل ذلك فإن لغة مواصفات سميت النيلين تم تطويرها بالإعتماد على المواصفات البديهية. أظهرت نتائج التقييم أن نظام التحقق المطور يزودنا بدرجة عالية من آلية الأثبات مقترنة مع قدرته على توفير تغذية راجعة في حالة محاولات الأثبات الفاشلة وهكذا يزيل الصعوبة المرتبطة بعملية تطبيق منطق هوار يدوياً.

TABLE OF CONTENTS

Dedication I

Acknowledgement II

Abstract III

مستخلص البحث IV

Table of Contents V

List of Tables XI

List of Figures XII

List of Abbreviations XIV

CHAPTER ONE INTRODUCTION 1

1.1 Introduction 1

1.2 Problem Statements and Its Significance 2

1.3 Research Questions 3

1.4 Research Hypothesis 3

1.5 Research Philosophy 3

1.6 Research Objectives 4

1.7 Research Scope 5

1.8 Organization of the Dissertation 5

CHAPTER TWO SOFTWARE ENGINEERING AND SOFTWARE QUALITIES 6

2.1 Introduction 6

2.2 The History of Software Engineering Discipline 7

2.3 Software Quality 11

2.3.1 Software Errors, Faults and Failures 11

2.3.2 Classification of the Causes of Software Errors 12

2.3.3 Software Quality Definition 14

2.3.4 Software Quality Attributes 16

2.3.4.1 Functional Attributes 16

2.3.4.2 Operational Attributes 19

2.3.4.3 Usability Attributes 21

2.3.4.4 Business Attributes 22

2.3.4.5 Structural Attributes 23

2.3.5 The Software Quality Challenges 24

2.3.6 Quality Assurance 27

2.3.6.1 A Classification Scheme 27

2.3.6.2 Defect Prevention 28

CHAPTER THREE SOFTWARE SPECIFICATION AND VALIDATION 33

3.1 Software Specifications 33

3.2 A Discipline of Specification 34

3.2.1 Specification: The Process and the Product 34

3.2.2 The Specification Product: Its Three Forms 35
3.2.3 Properties of Specifications 35
3.2.4 Data Types 36
3.3 Formal Specification 37
3.3.1 Classification of Formal Specification Methods 39
3.3.1.1 Property-Oriented Specification Methods 40
3.3.1.2 Model-Based Specification Techniques 41
3.4 The Specification of Data Types 41
3.4.1 A Relational Model 42
3.4.2 Axiomatic Representation 44
3.4.2.1 Specification of a Stack 44
3.4.2.2 Specification of a Queue 46
3.5 Specification Validation 48
3.5.1 ADT's Specification Validation 50
CHAPTER FOUR PROGRAM CORRECTNESS AND VERIFICATION 54
4.1 Introduction 54
4.2 Correctness of Programs 56
4.3 Software Verification 58
4.3.1 Hoare Logic 59
4.3.1.1 Hoare Triple 59

4.3.1.2 An Inference System 59

4.3.1.3 Illustrative Examples 63

4.3.2 Formal Software Verification Techniques 67

4.3.2.1 Static Analysis 68

4.3.2.2 Theorem Proving 72

CHAPTER FIVE AUTOMATED VERIFICATION WITH HAHA 74

5.1 Introduction 74

5.2 Tools for Formal Program Verification 75

5.2.1 The Key-Hoare Tool 75

5.2.1.1 Using KeY-Hoare Tool 76

5.2.2 Hoare Advanced Homework Assistant (HAHA) Tool 80

5.2.2.1 Using HAHA Tool 82

5.3 Evaluation 85

CHAPTER SIX ALNEELAIN SPECIFICATION LANGUAGE AND COMPILER 88

6.1 Introduction 88

6.2 Alneelain Specification Language and Compiler 89

6.2.1 Grammars and BNF for Alneelain Specification Language 91

6.2.2 Lexical Analysis 96

6.2.3 Syntax Analysis 97

6.2.3.1 Recursive Descent 98

6.3 The User Interface of Alneelain Specification Language 100

CHAPTER SEVEN VERIFYING ADT IMPLEMENTATIONS AGAINST AXIOMATIC SPECIFICATIONS 104

7.1 Introduction 104

7.2 Defining ADT's Specification 105

7.3 Implementation of Abstract Data Types 105

7.4 Mapping Axioms to Hoare Formulas 107

7.5 Verifying ADT Implementations Against Axioms 108

7.6 Submitting to HAHA Through the API 112

CHAPTER EIGHT IMPLEMENTATION AND DEPLOYMENT 121

8.1 Introduction 121

8.2 The Framework of the Verification Model 122

8.3 Software Deployment 125

8.3.1 QT Software 126

8.3.2 Microsoft Visual Studio 127

8.3.3 Spy++ 127

8.4 User Interface of Alneelain Verification System 128

CHAPTER NINE AUTOMATED VERIFICATION AND VALIDATION 131

9.1 Introduction 131

9.2 Examples of Correct Implementations that Succeed 133

9.3 The Assessment of Alneelain Verification System 136

CHAPTER TEN THE INTEGRATION 141

10.1 Introduction 141

10.2 Goal Oriented Testing 142

10.3 The User Interface of the Integrated Tool 147

10.4 The Assessment of Alneelain Verification and Testing System 149

CHAPTER ELEVEN CONCLUSION AND FUTURE WORK 152

11.1 Conclusion 152

11.2 Future Work 153

References 154

Appendix -A 162

Appendix -B 166

Appendix -C 180

Appendix -D 200

Appendix -E 210

LIST OF TABLES

Table No. **Page No.**

Table 2.1: Factors Affecting Defect Detection in Software Products vs. Other Industrial Products..26

Table 3.1: Two-Team, Two-Phase Approach..49

Table 6.1: Tokens, Lexemes, and Patterns for Alneelain Specifcation Language 97

Table 9.1: Modifications on Array-Based Stack Implementation and their Effects on Verification System 136

Table 9.2: Modifications on Array-Based Queue Implementation and their Effects on Verification System 137

Table 9.3: Modifications on Scalar-Based Implementation and their Effects on Verification System 137

Table 9.4: Modifications on Axioms of Stack Data Type and their Effects on Verification System 138

Table 9.5: Modifications on Axioms of Queue Data Type and their Effects on Verification System 139

Table 10.1: Modifications on Array-Based Stack Implementation and their Effects on Verification and Testing System..150

Table 10.2: Modifications on Scalar-Based Stack Implementation and their Effects on Verification and Testing System..150

LIST OF FIGURES

Figure No. Page No.

Figure 4.1: The Interpretation of Conditional Rule 61

Figure 4.2: The Interpretation of Alternation Rule 62

Figure 4.3: The Interpretation of Iteration Rule 63

Figure 5.1: Screen Shot of KeY- Hoare System 77

Figure 5.2: KeY-Hoare Input File for the Gauss Example 79

Figure 5.3: Screen Shot of HAHA System 81

Figure 5.4: HAHA Input File for the Gauss Example 83

Figure 6.1: The Flowchart for Steps of Designing Alneelain Specification Language ... 90

Figure 6.2: The BNF for Alneelain Specification Language 94

Figure 6.3: The Stack Specification in Alneelain Specification Language 95

Figure 6.4: Recursive Descent Parser for Grammar in Figure 6.2. 100

Figure 6.5: The User Interface of Alneelain Specification Language 101

Figure 7.1: The Implementation of Stack Data Type 107

Figure 8.1: The Verification Model 122

Figure 8.2: The Framework for the Verification Model 123

Figure 8.3: Proof Support Algorithm 125

Figure 8.4: The User Interface of Alneelain Verification System 130

Figure 9.1: Integer-Based Implementation of Stack Data Type 132

Figure 9.2: An Example for a Correct Array-Based Implementation for Stack 133

Figure 9.3: The Verification Report for Stack Implementation 134

Figure 9.4: An Example for a Correct Array-Based Implementation for Queue 135

Figure 9.5: An Example for a Correct Scalar-Based Implementation for Stack 135

Figure 10.1: The Test Driver 145

Figure 10.2: The Source Code of Pushpoprule() Function 146

Figure 10.3: The Testing Parameters Window 148

Figure 10.4: The Test Results of the Stack Data Type Implementation 149

LIST OF ABBREVIATIONS

GSEP	Good Software Engineering Practices
MTTF	Mean Time To Failure
MTBF	Mean Time Between Failures
MFC	Mean Failure Cost
MTTD	Mean Time to Detection
MTTE	Mean Time To Exploitation
QA	Quality Assurance
ADT's	Abstract Data Types
TAM	Trace Assertion Method
GCD	Greatest Common Divisor
BMC	Bounded Model Checking
HAHA	Hoare Advanced Homework Assistant
BNF	Backus-Naur Form
GUIs	Graphical User Interfaces

CHAPTER ONE
INTRODUCTION

1.1 INTRODUCTION

At the present time, the software is involved in almost all aspects of life, and largely and increasingly contributes to the world economy. Together with the rising demand for software to serve a wide range of needs, the world is also experiencing growing expectations in terms of product quality. The quality and the correctness of software are often the greatest concern in electronic systems.

Though modern computer systems consist of complex hardware and software components, ensuring the correctness of the software part is often a greater problem than that of the underlying hardware. As software, products take on increasingly critical functions in our global world, the verification of their correctness and their reliability grows increasingly critical. However, as they grow increasingly large and complex, the verification of their correctness is also growing increasingly difficult. This leaves a massive technological gap, which current software research seeks to fill. Despite several decades of research, the verification of industrial-size software products to adequate levels of precision and thoroughness remains an unfulfilled challenge. Methods developed in research laboratories do not scale up to industrial size problems, and techniques used in industrial practice fall short of the standards expected from the software industry.

We argue in favor of an eclectic approach that uses a variety of methods such as static analysis, dynamic testing, reliability estimation, fault tolerance, etc. [1] , where each method provides a good return on investment with some aspects of verification at the possible expense of other aspects. By virtue of the *Law of Diminishing Returns*, the application of such an approach enables us to maximize the impact of our verification effort. Specifically, we use one of the most common approaches to program verification, namely program verification using Hoare Logic [2], which is based on an axiomatic approach involving assertions for the verification to verify the correctness of Abstract Data Types (ADTs) implementations with respect to ADT's specifications

written in a trace-like notation [3]. This research is a part of a team effort that involves specification generation and validation, program verification, and testing.

1.2 PROBLEM STATEMENTS AND ITS SIGNIFICANCE

Computer systems correctness is vital in the contemporary information society. Despite the fact that modern computer systems are composed of complex hardware and software components, verifying the correctness of the software part is often a greater problem than that of the underlying hardware. It is known that manual inspection of complex software is error-prone and expensive, therefore tool support is required. There are many tools that attempt to discover design errors using test vectors by examining certain executions of a software system. Formal verification tools, on the other hand, are used to check the behavior of a software design for all input vectors. A large number of formal tools for checking functional design errors in hardware are available and widely used. In contrast, markets of tools that check software quality are still in a beginning stage.

According to practical experience, using static analysis methods for proving the correctness of computer programs is tedious, unreliable, obscure and entirely impractical process. This is because Hoare logic principles are generally not well understood. In addition, the whole concept is believed by a number of scholars as only a strange part of computer science history despite its fundamental role in the constantly developing field of program verification. No doubt that this critical state of affairs has numerous reasons. However, there is one reason or issue that has greatly led to this problem that is the verification tools used in applying Hoare logic. These tools, in many cases, consist of a pen and paper, making it a tedious task to verify a whole program using a pen and a sheet of paper. It is more difficult than checking the correctness of the very same program in a less formal way. Even worse, both approaches are considered to have similar chances of making a mistake. Performing the whole logical inference on paper, without using the computer (save, perhaps, in the matter of typesetting) supports the view that static analysis is an impractical verification tool. A direct way to solve this problem is to use automated formal verification software to facilitate checking the correctness of Hoare programs.

1.3 RESEARCH QUESTIONS

The main question that will be addressed in this research is how to build an automated tool that attempts to verify the correctness of Abstract Data Type implementation against the axioms of specification.
There are additional sub-questions as follows:

1. What is an appropriate methodology to write the ADT's specification?
2. What is the suitable technique for verifying ADT's implementation?
3. How does the proposed approach compare with current approaches based on Z, B, Alloy, etc.…?

1.4 RESEARCH HYPOTHESIS

Precise abstract software specification is achievable by using formal specification languages. However, nontrivial specifications are extremely difficult to read and write. An axiomatic specification proposed by [4] is used as a specification language to write the abstract data type specification in this research. Even though this specification model was developed independently, it bears much resemblance to the trace specifications [3]. It introduces the data type by means of behavioral formal specifications, that describe the functional attributes of the data type, but give no insight into how the data type may be implemented; specifications are represented by axioms and rules, where the axioms reflect the behavior of the data type for simple data values and rules define their behavior inductively for more complex data values. To verify that the ADT's implementation satisfies the axioms of the specification, the Hoare logic is used. Hoare Logic is a *compositional proof technique* for proving that the implementation meeting the specifications— whereby "compositional" we mean that proving the correctness of a complex program can be decomposed into proofs of the correctness of its components. It lies at the core of a huge variety of tools that are now being used to verify real software systems.

1.5 RESEARCH PHILOSOPHY

Various approaches have been proposed in the literature for the specification of abstract data types. The axiomatic specification is considered to be an important

specification methodology because it satisfies the simplicity, formality, and abstraction criteria. It is easy to become familiar with the inductive arguments that are needed to build axiomatic specifications. Also using the axiomatic notation makes the specification validation easier and effective.

The Hoare Logic, which is an axiomatic approach to proving program correctness, has great consequence in the methods for verifying programs. Hoare logic introduced the strength of formal logic in computer programming, not only as a tool to reason about program properties but also to derive programs from their specifications. It is precise; clearly delineates the "what" with the "how" basis for most other verification methods and including semiformal specification notations. What makes Hoare logic difficult to apply on a large scale, and in particular, what makes it difficult to automate is the need to invent invariant assertions for iterative constructs in programs. However, our approach obviates this obstacle because the axioms represent simple basic behavior of the ADT, they do not involve complex calculations in the source code; at most, they may involve some loops to initialize a data structure. Such code either uses loops for simple tasks, for which invariant assertions are simple to find, or does not invoke loops at all.

1.6 RESEARCH OBJECTIVES

The main objective of this research is to build an automated tool that takes the verification conditions (in the form of a Hoare formula) associated to each axioms of ADT's specification and attempts to prove them in Hoare's logic, using a verification tool such as Hoare Advanced Homework Assistant (HAHA). Specific objectives are listed as follows:

1. To investigate variants of techniques those have been applied to formal software specification and verification.
2. To specify ADT's in a behavioral style using an axiomatic specification.
3. To verify ADT's implementation against axiomatic specification using Hoare Logic.
4. To evaluate the performance of our tool against the performance of others whenever applicable.

1.7 RESEARCH SCOPE

This research is concerned with verifying ADT’s implementation against axiomatic specification, for that we concerned only by the axioms of the specification and exclude the rules. The tool verify any software product whose specification can be written using the trace notation and the axiomatic representation. This includes most (if not all) ADT's as well as any product whose specification can be written in our specification model. The tool is highly automated and do not require substantial manual effort or user interaction.

1.8 ORGANIZATION OF THE DISSERTATION

This dissertation consists of eleven chapters. The first one presents the introduction, problem statements and its significance, research questions, research hypothesis, research philosophy, research objectives and research scope.

Chapter 2: reviews the literature on software engineering and software qualities.

Chapter 3: covers software specification and validation

Chapter 4: dives into program correctness and verification

Chapter 5: is devoted to automated verification with HAHA tool.

Chapter 6: presents the design of a new specification language called Alneelain specification language.

Chapter 7: is devoted to verification of ADT implementations against axiomatic specifications.

Chapter 8: presents how we implement and deploy Alneelain verification system.

Chapter 9: introduces automated verification using Alneelain verification system and the assessment of the system

Chapter 10: explains the integration of Alneelain verification system with a testing system.

Finally, chapter eleven represents the overall conclusions in this dissertation and research with it is orientation for future work.

CHAPTER TWO
SOFTWARE ENGINEERING AND SOFTWARE QUALITIES

2.1 INTRODUCTION

Intuitively, when one thinks about software, one imagines an accumulation of programming language instructions and statements or development tool instructions that together make up or constitute a program or software package. This program or software package is usually termed the “code”. IEEE defines software as [5]:

“Software is computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system”.

The IEEE definition of software enumerates four components of the software: Computer programs (the code), Procedures, Documentation and Data necessary for operating the software system. All these components are required for assuring the quality of the software development process and for future maintenance services [6].
At the present time, the software is involved in almost all aspects of life, and largely and increasingly contributes to the world economy. This growing share of the software began with a slow pace together with the invention of computing in the twentieth century, and was further accelerated by the appearance World Wide Web at the end of the twentieth and the beginning of the twenty first century. This phenomenon has created a rising demand for software products and services, and the software industry faced a huge market pressure to supply demanded products and services [7].
A large number of science and engineering fields including Bioinformatics, medical informatics, weather forecasting, modeling and simulation, etc. greatly rely on software to the extent that they are looked at as pure applications of software engineering. Also, it is widely observed that curricula in computer science are gradually heading towards increasing software engineering contents replacing traditional theoretical material that are considered to be irrelevant to the current job market. The rising share of software made several engineering colleges start awarding software engineering degrees in computer science departments or starting complete software engineering departments alongside traditional computer science departments [7].

Software engineering deals with all aspects of software production starting from the initial stages of system specification through to maintaining the system after its application and utility. It's defined by IEEE [5] as:

“Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software”.

Actually, software engineering was first intended to attain these purposes: improving the quality of software, accomplishing projects exceeding time and budget under control, and ensuring that software is built systematically, rigorously, measurably, on time, on budget, and within specification [8]. Engineering already addresses all these issues, hence the same principles used in engineering can be applied to software.

Together with the rising demand for software to serve a wide range of needs, the world is also experiencing growing expectations in terms of product quality. As software takes on ever extremely critical functions in life-critical and mission-critical applications, and in applications that perform massive financial tasks, it is very important to make sure that software products perform their functions with a high level of reliability. This requires the deployment of different kinds of techniques, including: process controls which ensure that software products are developed and evolved according to certified, mature processes; and product controls which ensure that software products meet quality standards commensurate with their application domain requirements. This is usually accomplished by combining techniques such as static analysis, dynamic testing, reliability estimation, fault tolerance, etc. [7].

In short, it is acceptable to state that software industry is under massive pressure to provide both quantity and quality, which is both difficult and expensive.

2.2 THE HISTORY OF SOFTWARE ENGINEERING DISCIPLINE

Apparently, software engineering is similar to an engineering discipline like chemical engineering, mechanical engineering, civil engineering, electrical engineering, etc. However, software engineering differs from other engineering disciplines in many ways [7].

Not like civil engineering and mechanical engineering, which date back to antiquity or chemical engineering and electrical engineering, which date back to the eighteenth

century, software engineering emerged in the second half of the twentieth century. It is a relatively a younger discipline. According to Mili and Tchier [7] the brief history of the software engineering discipline can be divided into five decades or eras:

1. **The Sixties: The Era of Pioneers.** During this decade or era, practitioners and researchers came face to face for the first time with the complexities, paradoxes, and anomalies of software engineering. Software projects conducted during this era were attempts in the unknown that have high levels of risk, unpredictable outcomes, and massive cost and schedule overruns. Limited number of programming languages were used during this era, namely: assembler, FORTRAN, COBOL and (in academia) Algol.

2. **The Seventies: Structured Software Engineering.** This decade or era was dominated by the general belief that software engineering problems were of a technical nature, and that all software engineering problems would be resolved if scholars devised techniques for software specification, design and verification. Given that structure was the scholars' main intellectual tool for dealing with complexity, this era has witnessed the development of various structured techniques such as structured programming, structured design, structured analysis, structured specifications, etc. Programming languages utilized during this decade were dominant C and (in academia) Pascal.

3. **The Eighties: Knowledge-Based Software Engineering.** During this decade or era, it was generally realized that software engineering problems were of a managerial and organizational nature rather than a technical nature. This realization came together with the emergence of the Fifth Generation Computing Initiative. The initiative started in Japan and then spread to the US, Europe, Canada and the rest of the world and it focused on thinking machines designed with extensive use of artificial intelligence techniques. This general approach permeated the discipline of software engineering with the emergence of knowledge-based software engineering techniques. The programming languages that were predominantly used during this era were: Prolog, Scheme/Lisp and Ada.

4. **The Nineties: Reuse-Based Software Engineering.** Because the fifth generation computing initiative failed to accomplish its objectives, and it started to fade worldwide, software researchers and practitioners focused on reuse as a new

alternative technique for the discipline. It was known in software engineering discipline, reuse was not an integral part of the routine engineering process. It was thought that if only software engineers had large databases of reusable software components readily available, the industry would achieve great gains in productivity, quality, time to market, and reduced process risk. This evolution came together with the emergence of object-oriented programming, which supports a bottom design discipline that facilitates product reuse. The programming languages that were predominantly used during this era were: C, C++, Eiffel and Smalltalk.

5. **The First Decade of the Millennium: Lightweight Software Engineering.** While software reuse was not practically a general paradigm in software engineering, it was feasible in limited application domains, giving rise to *product line engineering*. Other attributes of this era included Java programming with its focus on web applications; agile programming, with its focus on rapid and flexible response to change; component-based software engineering with its focus on software architecture and software composition. The programming languages that were predominantly used in this era were: Java, C++, and (in academia) Python.

Probably as a result of this short but eventful history, the discipline of software engineering has a number of paradoxes and counterintuitive properties as explained in [7], these are:

1. **Large, Complex Products**

   Not only is it critical for software engineers to build software products that are of high quality, it is also very difficult, due to their size and complexity.

2. **Expensive Products**

   Not only are software products very large and complex, they are also very expensive to produce.

3. **Absence of Reuse Practice**

   In the absence of economies of scale, it is hoped that costs will be controlled by a routine discipline of reuse; however, in the case of software, it turns out that reuse is very difficult to achieve on a routine basis.

4. **Fault Prone Designs**

   Since the design space of software products is broader than that of any other engineering products, it is much more difficult to codify and standardize best practices in software engineering than it is in any other engineering discipline.

5. **A Labor Intensive Industry**

   In other engineering disciplines most of the cost (more than 99%, perhaps) comes from manufacturing rather than from the design process. By contrast, when one buys a software product, he/she pays essentially for the design effort, as there are no manufacturing costs to speak of (loading compact disks or downloading program files).

6. **Absence of Automation**

   The labor-intensive nature of software engineering has a direct effect on the possibility to automate software engineering processes. In all engineering processes, it is possible to achieve savings in manufacturing by automating the manufacturing process or at least streamlining it, as in assembly lines. This is possible since manufacturing follows a simple, systematic process that requires little or no creativity. By contrast, it is not possible to automate design. This is because design requires intensive human input including creativity, artistic appreciation, aesthetic sense, etc.

7. **Limited Quality Control**

   The lack of automation, hence the absence of process control, makes the control of product quality very difficult. Comparatively, in other engineering disciplines the production process is usually a systematic, repeatable process and quality control can analytically be performed by certifying the process, or empirically by statistical observation. However, the production of software goes through a creative labor intensive process, therefore both approaches are not feasible, since such creative process is neither systematic nor repeatable. This changes the control of product quality to product controls including static analysis, or dynamic program testing.

8. **Unbalanced Lifecycle Costs**

   In most other engineering disciplines, products are massively produced and are in general is assumed to have an expected behavior. In software engineering, due to arguments mentioned above, assumption of expected behavior is unfounded.

Therefore, the sole method to guarantee the quality of a software product is to subject that product to extensive analysis. Practically, this is discovered to be an expensive proposition and a source of another massive paradox in software engineering economics. Also, in other engineering manufacturing, testing (and, more generally, verification and quality assurance) constitutes a small percentage of the production cost. By contrast, testing accounts for a large percentage of the lifecycle cost of a software product.

9. **Unbalanced Maintenance Costs**

   In software maintenance, it is common to distinguish several types of activities. In terms of cost the two most important types of these activities are:

   1. Corrective maintenance that aims at removing software faults.
   2. Adaptive maintenance that aims at adapting the software product to evolving requirements.

   Empirically, studies show that adaptive maintenance represents the vast majority of maintenance costs. In contrast, in other engineering disciplines, there is almost no adaptive maintenance to be mentioned.

2.3 SOFTWARE QUALITY

When quality or high-quality is associated with a software system, it is an indication that few or no software problems are expected to be discovered during its operations. In addition, when problems occur, they are expected to have negative minimal impact. Related issues are discussed in this section.

2.3.1 SOFTWARE ERRORS, FAULTS AND FAILURES

Key to the correctness aspect of software quality is the concept of defect, failure, fault, and error. The term *defect* generally refers to some problem with the software, either with its external behavior or with its internal characteristics. The IEEE [5] defines the following terms related to defects:

1. *Failure:* The inability of a system or component to perform its required functions within specified performance requirements.
2. *Fault:* An incorrect step, process, or data definition in a computer program.
3. *Error:* A human action that produces an incorrect result.

Therefore, the term *failure* refers to a behavioral deviation from the user requirement or the product specification; *fault* refers to an underlying condition within a software that causes certain failure(s) to occur; while *error* refers to a missing or incorrect human action resulting in certain fault(s) being injected into a software.

Also, errors can be extended to include *error sources,* or the root causes for the missing or incorrect actions, such as human misconceptions, misunderstandings, etc. Failures, faults, and errors are collectively referred to as *defects* in literature [9].

Originally, software failures occur because of software errors made by programmers. Such an error is different in nature. It can be a grammatical error in one or more of the code lines, or a logical error in carrying out one or more of the client's requirements. However, not all software errors become software faults. In some cases and in either general or specific applications, a software error may lead to improper functioning of the software. In many other cases, erroneous code lines will affect the functionality of parts of the software and not its whole functionality; in a part of these cases, such errors will be corrected or masked by subsequent code lines [6].

Here, the concern is mainly with software failures that disrupt the use of that software. The need is to examine the relationship between software faults and software failures. As known, not necessarily that all software faults end with software failures. A software fault becomes a software failure only when it is operated or activated by users. In several cases, a software fault is never activated because users are not interested to use that specific faulty application or because the combination of conditions necessary to activate the fault never occurs. In many cases, only a portion of the software faults, and in some cases only a small portion of them, will turn into software failures in either the early or later stages of the software's application. Other software faults will remain hidden, invisible to the software users, yet capable of being activated when the situation changes. Importantly, developers and users have different views of the software product regarding its internal defects. While developers are interested in software errors and faults, their elimination, and the ways to prevent their generation, software users are worried about software failures [6].

2.3.2 CLASSIFICATION OF THE CAUSES OF SOFTWARE ERRORS

As software errors usually result in poor software quality, it therefore is important to investigate the causes of these errors in order to prevent them. Such

errors are of different types including code errors, procedure errors, documentation errors, or data errors. It should be emphasized that the causes of all these errors are human. They are made by systems analysts, programmers, software testers, documentation experts, managers and sometimes clients and their representatives. In rare cases, software errors may be due to the development environment (interpreters, wizards, automatic software generators, etc.). Therefore, it is reasonable to claim that in almost all cases human error causes the failure of the development environment tool. According to Galin [6], causes of software errors can be further classified according to the stages of the software development process in which they occur, as follows:

1. **Faulty Definition of Requirements**

   The faulty definition of requirements is one of the main causes of software errors. It is an error made by clients.

2. **Client–Developer Communication Failures**

   Misunderstandings resulting from improper client–developer communication are the main causes for the errors that prevail in the early stages of the software development process.

3. **Deliberate Deviations From Software Requirements**

   In numerous cases, developers deliberately deviate from the documented requirements. Such deliberate deviations often cause software errors, which are mainly byproducts of the changes.

4. **Logical Design Errors**

   When formulating software requirements, professionals who design the system – systems architects, software engineers, analysts, etc. – cause logical design errors to enter the system.

5. **Coding Errors**

   Programmers make coding errors due to a number of reasons including; misunderstanding the design documentation, linguistic errors in the programming languages, errors in the application of CASE and other development tools, errors in data selection, and so forth.

6. **Non-Compliance with Documentation and Coding Instructions**

   It is well-known that almost every development unit keeps its own documentation and coding standards in order to define the content, order and format of the

documents, and the code created by team members. In addition, the unit develops and publicizes its templates and coding instructions so that members of the development team will be able to comply with these requirements. Non-compliance with these standards set by the unit will cause software errors. The reasons for such errors is that the quality risks of non-compliance result from the special characteristics of the software development environment. Even if the quality of the "non-complying" software is acceptable, the future handling of this software (by the development and/or maintenance teams) is expected to increase the rate of errors.

7. **Shortcomings of the Testing Process**

   Shortcomings or weakness of the testing process increases the error rate by leaving a greater number of errors undetected or uncorrected.

8. **Procedure Errors**

   Procedures are intended to direct the user regarding the activities required at each step of the process. They are particularly important in complex software systems where processing is conducted in several steps, each of which may feed a variety of types of data and allow for examination of the intermediate results. Failure to properly follow the procedures results in errors, called procedure errors.

9. **Documentation Errors**

   The documentation errors that affect the development and maintenance processes are errors encountered in the design documents and in the documentation integrated into the body of the software. They usually cause additional errors in further stages of development and also during maintenance. In addition, documentation errors found in the user manual affect mainly software users, and may appear in the "help" displays incorporated in the software.

2.3.3 SOFTWARE QUALITY DEFINITION

The previous discussion on software components, errors and their causes, and knowledge that errors harm the quality of the software, have set the scene to define software quality. IEEE defined software quality as [5]:

1. "The degree to which a system, component, or process meets specified requirements".

2. "The degree to which a system, component, or process meets customer or user needs or expectations".

There are two alternative definitions of software quality, held by the founders of modern quality assurance, Philip B. Crosby and Joseph M. Juran. Each definition reflects a different conception of software quality:

1. "Quality means conformance to requirements" [10].
2. "Quality consists of those product features which meet the needs of customers and thereby provide product satisfaction. Quality consists of freedom from deficiencies" [11].

Crosby's definition of software quality refers to the extent to which the written software satisfies the specifications prepared by both the customer and the professional team. His definition implies that errors included in the software specification are not considered and assumed to have no effect on the software quality. Such implied characteristic is considered as the approach's deficiency [6]. Juran's definition aims at achieving customer satisfaction, and views the satisfaction of customers' real needs as the ultimate goal of software quality. The adoption of the second definition requires the developer to exert significant professional efforts in examining and correcting the customer's requirements specifications. One main deficiency of this definition is that customers are not professionally responsible for the accuracy and completeness of the software specifications. Also, following this conception, the customer is able to state his real needs. Such needs may not agree with the project specifications on one or more issues, at a very late stage of the project, even at the final stage [6]. Consequently, difficulties will occur during the development process of the project, especially when trying to demonstrate how well the program satisfies the user's needs. However, aspects of software quality are evidently included in the definition suggested by Pressman [12] who defined software quality as:

"Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software".

It is clear that Pressman's definition suggests three requirements for quality assurance that are to be met by every developer [6]:

1. Specific functional requirements i.e. the outputs of the software system.
2. The software quality standards stated in the contract.
3. Good Software Engineering Practices (GSEP), reflecting state-of-the-art professional practices, to be met by the developer even though not explicitly mentioned in the contract.

Effectively, Pressman's definition includes operative directions for testing the degree to which the requirements are met.

2.3.4 SOFTWARE QUALITY ATTRIBUTES

The software product may feature several quality attributes. These attributes are briefly reviewed here, focusing on those dealing with software testing. According to Mili and Tchier [7] software quality attributes can be divided broadly into three categories:

1. *Attributes relevant to software users*: These include functional attributes, operational attributes, and usability attributes. They reflect the overall quality of service delivered by the software product.
2. *Attributes relevant to software operators*: These include business attributes that reflect the cost of developing, maintaining and evolving the software product.
3. *Attributes relevant to software engineers*: These include structural attributes that reflect the engineering qualities of the product. They have impact on analyzing and working with the software product for the entire duration of its lifecycle.

This classification is not perfectly orthogonal as many attributes may lie within more than one category; however, it helps to define some structure in the set of attributes. These families of attributes are discussed in turn, in this section.

2.3.4.1 FUNCTIONAL ATTRIBUTES

Functional attributes characterize the input/output behavior of software products. Two broad categories of functional attributes are distinguished here: those

with a Boolean nature (a software product has them or does not have them) and those that are of a statistical nature (a software product has them to a smaller or larger extent). They depend on the existence of a specification, which describes a set of situations the product is intended to face, along with a prescription of correct program behaviors for each situation. The set of relevant situations is referred to as the *domain* of the specification.

1. **Boolean Attributes**

   There are two attributes of a Boolean nature in a software product:

   1. *Correctness*, i.e. the software product behaves according to its specification for all possible situations in the domain of the specification.
   2. *Robustness*, i.e. the software product behaves according to its specification for all possible situations in the domain of the specification, and behaves *reasonably* for situations outside the domain of the specification. Obviously, reasonable behavior is not a well-defined condition, hence robustness is only partially defined; however, it generally refers to behavior that alerts the user to the abnormality of the situation, and acts carefully and conservatively i.e. avoiding irreversible operations, avoiding irretrievable losses of information, etc.

   From its definition, robustness logically implies correctness: whereas correctness refers solely to the behavior of the software product within the domain of the specification. Robustness attribute also refers to the behavior of the product outside the domain of the specification. Conversely, one can argue that robustness is not distinguishable from correctness, since it is merely correctness with respect to a stronger specification (one that specifies the behavior of candidate programs inside the original specification's domain, as well as outside it); nevertheless, for a given specification, robustness and correctness are distinct properties.

2. **Statistical Attributes**

   Correctness and a fortiori robustness are notoriously difficult to establish for software products of any realistic size. As a result, statistical attributes are introduced. They measure (over a continuum) how close a software product is to being correct or robust. There are two broad families of statistical attributes, based on whether the obstacles to correctness and robustness are inadvertent (product

complexity, programmer incompetence, unexpected circumstance, etc.) or voluntary (malicious attempts to cause product failure).

1. *Dependability*. i.e. the probability that the system behaves according to its specifications for a period of operation time. Two attributes are recognized within dependability: *reliability* and *safety* that differ by the stakes attached to satisfying the specification.
   (a) *Reliability* i.e. the probability that the software product operates for a given amount of time without violating its specification.
   (b) *Safety* i.e. the probability that the software product operates for a given amount of time without causing a catastrophic failure.

Both reliability and safety mean that the product is able to operate according to its specification. However, reliability means the product's ability to operate according to all the clauses of its specification, while safety is specifically concerned high-stakes clauses. If a product violates these clauses, it will be subject to a catastrophic loss, in terms of human lives, mission success, high financial stakes, etc. An alternative term for "safe systems" is "fail-safe", meaning that it may fail to satisfy its specification, but at the same time satisfies the high-stakes requirements of its specification. As a result, a system may be reliable but unsafe (i.e. fails seldom, but causes catastrophic consequences whenever it fails); and a system may be safe but unreliable (i.e. fails often, but causes low stakes losses, never causes catastrophic losses).

A well-known and frequently used metric to quantify reliability is the "Mean Time To Failure (MTTF)", defined as the mean of the random variable that represents the operation time until the next system failure; MTTF can also be used to quantify safety, if Failure is just replaced by Catastrophic Failure. The Mean Time Between Failures (MTBF) is an older metric, defined as the random variable that represents the time between two successive failures. The Mean Failure Cost (MFC) is a more recent metric, intended to measure the mean of the random variable that measures the loss of a stakeholder resulting from possible system failures.

2. *Security*. Whereas dependability refers to failures that result from poor system design, security refers primarily to intended or deliberate actions by malicious

perpetrators. Although it could be argued that, these actions become possible due to system vulnerabilities, which originally stem from poor system design. Four attributes can be considered as aspects of security:

(a) *Confidentiality.* It is the system's ability to prevent unauthorized access to confidential data entrusted to its custody.

(b) *Integrity.* It is the system's ability to prevent loss or damage to critical data entrusted to its custody.

(c) *Authentication.* It is the system's ability to properly identify each user that gains access to its resources, and to grant users access privileges according to their status.

(d) *Availability.* It is the system's ability to continue delivering service to its user community; it is normally expressed as a percentage. This attribute usually causes denial of service attacks i.e. when the system is under attack; its ability to deliver services to its legitimate users suffers.

There is no commonly accepted measure of system security. Since security attacks result from system vulnerabilities, quantifying all dimensions of security (including availability) is possible by using the Mean Time to Detection (MTTD). MTTD is the mean of the random variable that measures the time it takes perpetrators to uncover system vulnerabilities. Quantification of security is also possible by Mean Time To Exploitation (MTTE) i.e. the mean of the random variable that represents the time it takes perpetrators to find a way to exploit system vulnerabilities.

2.3.4.2 OPERATIONAL ATTRIBUTES

Operational attributes are different from functional attributes, in the sense that functional attributes characterize the functions/ services that a software product delivers to its users, and operational attributes characterize the circumstances under which these services are delivered. Five operational attributes are presented below:

1. *Latency*. Latency (or response time) is defined as the time (measured in seconds) that elapses between the submission of a query to the system and the response to the query; this attribute is relevant for interactive systems. Latency usually varies from query to query, and its length is affected by the system workload at the time

the query is submitted. Therefore, it is usually expressed as an average over many queries, for an average workload.

2. *Throughput*. Throughput is defined as the volume of processing that the system can deliver per unit of operation time; this attribute is relevant for batch systems, such as the program that a bank runs in the middle of the night to update all the transactions of the day, and is usually measured in transactions per second. Latency and throughput attributes are related. This is because both attributes reflect the speed with which systems are capable to process transactions. The two attributes are fairly orthogonal i.e. a system can have short (good) latency and small (bad) throughput (if it spends much of its time switching between queries), and long (bad) latency and large (good) throughput (if its scheduler favors keeping the processor busy over the concern of fairness).
3. *Efficiency.* This operational attribute of a software product is usually defined as the software ability to deliver its functions and services with minimum amount of computing resources. Such resources include CPU cycles, memory space, disk space, network bandwidth, etc. It is difficult to quantify efficiency for a given software product in a way that prorates resource consumption to the services provided by that product. But, it is easy to use efficiency for comparing numerous software products for a given application.
4. *Capacity*. The capacity attribute is defined as the number of simultaneous users that a system can sustain while preserving a degree of quality of service (in terms of response time, timeliness, precision, size of data, etc.). This definition is unclear in numerous ways. For instance, what does it means to be simultaneous? At what level of granularity are we defining simultaneity? What degree of quality are we considering? etc. However, capacity remains a useful metric.
5. *Scalability*. This attribute reflects the system ability to keep delivering adequate service even when its workload exceeds original capacity. Scalability is sometimes referred to as “graceful degradation”: it is expected that when workload increases beyond the system original capacity, there will be some degradation of the service quality; scalability consists in ensuring that this degradation remains graceful (i.e. a progressive, continuous function of the workload).

2.3.4.3 USABILITY ATTRIBUTES

Usability attributes are different from functional attributes and operational attributes in the sense that they reflect the extent to which a software product can easily be used, and be customized to suit the needs and circumstances of the potential end users. Five usability attributes are presented and discussed below:

1. *Ease of Use*. Ease of use is an important attribute especially in systems intended for broad, diverse, heterogeneous and possibly unskilled user community. Qualities that support ease of use include: simplicity of system interactions; availability of help menus; use of simple vocabulary; tolerance to misuse; etc.

2. *Ease of Learning*. Ease of learning is an important attribute particularly in systems that have homogeneous user community in terms of skill level and who will use the software product for a long time period. Qualities that support ease of learning include: intuitiveness of system interactions, consistency of interaction protocols, uniformity of system outputs, etc.

3. *Customizability*. The customizability of a software product means its ability to be tuned to specific functional requirements of the end user, by the end user. The more control the end user has over the functionality of the software product, the better the customizability.

4. *Calibrability*. The calibrability of a software product is its ability to be tuned to specific operational requirements of the end user, by the end user. The more control the end user has over the operational attributes of the software product, the better the calibrability.

5. *Interoperability*. The interoperability attribute means the ability of a software to operate in conjunction with other applications; generally, there is no quantitative metric for this attribute. However, interoperability of a software can be qualitatively assessed by the range of applications with which it can operate (for example, the breadth of file formats it can analyze and process, or by the range of file formats in which it can produce its output).

These three attributes of a software product (functional attributes, operational attributes, and usability attributes) are of great interest to the end user.

In the rest of this section, other stakeholders of a software product are considered, and attributes that are of interest to them will be reviewed.

2.3.4.4 BUSINESS ATTRIBUTES

Business attributes reflect the stakes of a software manager in the development, operation and evolution of a software product. They are listed and discussed below:

1. *Development Cost*. It is an important business attribute that is usually quantified in person-months invested in the development of a software product, from its requirements analysis to its acceptance testing.
2. *Maintainability*. The maintainability of a software product is defined as how much effort is required for the maintenance of that product during the operation phase (post-delivery). Maintainability can either be quantified in absolute terms (person-months per year) or in relative terms (person-months par year per line of code). It is noticed that most maintenance costs are allocated to adaptive maintenance. This type of maintenance cost is usually driven by end user requirements rather than by any intrinsic attribute of the software product. Therefore the measure of maintainability to the volume of adaptive maintenance is normalized, or exclude adaptive maintenance altogether from the person-month calculation of maintenance effort.
3. *Portability*. The portability of a software product is defined as the average cost of porting that product from one hardware/ software platform to another. Enhancing portability is usually attained by reducing the platform-dependent functionality to the minimum level and by confining it to a single component of the product (hence confining changes to this component in the event of a migration of the product from one platform to another). Portability is not a purely qualitative attribute since it is possible to quantify it by the average cost (in person-months) of a migration of a software from one platform to another.
4. *Reusability*. Unlike the above-mentioned three attributes that measure product cost, reusability attribute reflects potential benefits of a software product. It is defined as the software product ability to be reused, in whole or in part, in the design and development of other software products within the product's application domain. Reusability comprises two orthogonal properties, namely:
   (a) *Usefulness*, defined as the extent to which a software product (or a component) is widely needed in the product's application domain.

(b) *Usability*, defined as the ease with which it is possible to adapt a software product (or a component) to the requirements of an application within the domain.

It is important to note that usefulness is a property of the software product's (or component's) specification while usability is a property of its design. Well-designed components can be adapted to related requirements at low cost, by such devices as genericity, parameterization, etc.

2.3.4.5 STRUCTURAL ATTRIBUTES

Structural attributes are of interest to these stakeholders: system custodians/ developers/ maintainers/ operators. Business attributes deal with the economic aspects of system management, structural attributes deal with their technical aspects. In other words, whereas business attributes are relevant to software managers, structural attributes are of interest to engineers, designers and other technical personnel. Four structural attributes are discussed below:

1. *Design Integrity*. It is easier to recognize the quality of a design than to define it, or to quantify it. High quality design should comprise the following features: orthogonality, economy of concept, the cohesiveness of the design rationale, consistency of design rules, adherence to a simple design discipline, etc.

2. *Modularity*. Modularity comprises "information hiding" as an important single design principle. This means that each component of the system hides a design decision that other components need not know about (i.e. be influenced by). Therefore, one important feature of a modular design is the separation between the specification of a module and its implementation. It is possible to quantify modularity by two attributes, these are:

   (a) *Cohesion*. It is defined as the volume of information flow within a component, and can be quantified using information theory.

   (b) *Coupling*. It is defined as the bandwidth of information interchange that takes place between components. It is possible to quantify coupling by the entropy of the random variable that represents the flow of information interchange.

3. *Testability*. It is the extent to which a system or its components can be tested to an arbitrary level of thoroughness. Two attributes are used to quantify testability:

(a) *Controllability*. The controllability of a component in a system is defined as the bandwidth (breadth) of input values that can be submitted (as test data) to the component by controlling system inputs. Controllability can be quantified by the condition entropy of the system's input given an input value of the component.

(b) *Observability*. The observability of a component is defined as the extent to which one can infer the output produced by the component by observing the system output. Observability is usually quantified by the conditional entropy of the component's output, given the system's output.

4. *Adaptability*. The adaptability of a system is defined as the ease with which it can be modified to satisfy changing requirements. Adaptability is somewhat similar to customizability. Both attributes deal with adjusting the software product to meet different sets of requirements. However, they are actually different in two crucial ways: while customizability is concerned with, changes made by the end user (hence accessible to her/him, by design), adaptability is concerned with changes made by the software engineer. Also, while customizability is concerned with changes that are planned for within the design of the system, adaptability is concerned with changes in the system requirements.

In many cases, structural attributes are found to support business attributes. An example of this is that modularity supports maintainability. However, they are considered as distinct attributes, in the sense that one is a business attribute while the other is a structural/ technical attribute. In addition, no one-to-one correspondence between them exists: modularity supports not only maintainability but also development cost (which it reduces); also, maintainability is supported not only by modularity but also by testability.

2.3.5 THE SOFTWARE QUALITY CHALLENGES

Actually, there are basic principal differences between software and other industrial products, such as automobiles, washing machines or radios. These differences are listed and discussed below [6]:

1. *Product complexity*. Product complexity is usually assessed by the number of operational modes the product permits. Considering their different settings,

industrial products do not permit more than a few thousand operational modes. In the case of software products, millions operational possibilities can be detected. It is a great challenge to software industry to ensure that this higher number of operational possibilities is correctly defined and developed.

2. *Product visibility*. Another evident distinguishing difference between the two types of products is that while industrial products are visible, software products are invisible. Visibility enables detecting defects in an industrial product during the manufacturing process. It is obvious that missing a part in an industrial product will be highly visible (for example a door missing from your new car). In contrast, defects in software products (whether stored on diskettes or CDs) are invisible, as is the fact that parts of a software package may be absent from the beginning.
3. *Product development and production process*. The phases at which the possibility of detecting defects in an industrial product are numerous, they include:
   (a) *Product development*. In this phase defects may be detected by designers and Quality Assurance (QA) staff who check and test the product prototype.
   (b) *Product production planning*. The production process and tools are designed and prepared during this phase. For example, a special production line may be designed and built to manufacture some products. This phase provides additional opportunities to inspect the product, which may reveal defects that "escaped" the reviews and tests conducted during the development phase.
   (c) *Manufacturing*. The application of QA procedures during this phase makes it possible to detect defects of the products. Detected defects in the first period of manufacturing can usually be corrected. This could be done by a change in the product's design or materials or by changing the production tools, in a way that eliminates such defects in products manufactured in the future.

Comparatively, defects in software products cannot be detected during all three phases of the production process (development, planning and manufacturing).The only phase when defects in software products can be detected is the development phase. Below is a review of what each phase contributes to the detection of defects in software products:

(a) *Product development*. During this phase, development teams and software quality assurance professionals direct their effort toward detecting inherent product

defects. At the end of this phase, they usually approve a product prototype that is ready for reproduction.

(b) *Product production planning*. In software production, process there is no need for planning phase. The manufacturing of software copies together with printing user's manual are conducted automatically. This applies to any software product, regardless of the number of copies produced.

(c) *Manufacturing*. As stated above, the manufacturing of the software product is restricted to copying the product and printing copies of the software manuals. Consequently, detecting defects during this phase is quite limited.

The differences affecting the detection of defects in software products compared to other industrial products are shown in Table 2.1 [6].

Table 2.1: Factors Affecting Defect Detection in Software Products vs. Other Industrial Products

Characteristic	Software Products	Other Industrial Products
Complexity	Usually, very complex product allowing for a very large number of operational options.	Degree of complexity much lower, allowing a most a few thousand operational options.
Visibility of product	Invisible product, impossible to detect defects or omissions by sight(e.g. Of a diskette or CD storing the software).	Visible product, allowing effective detection of defects by sight.
Nature of development and production process	Opportunities to detect defects arise in only one phase, namely product development.	Opportunities to detect defects arise in all phases of development and production: - Product development - Product production - Planning - Manufacturing

The fundamental differences between the development and production processes in software products and in industrial products result in creating different QA methodology for software products. Special tools and methods are devised for the software industry. Such tools and methods are well-documented in the professional publications and stated in special standards devoted to QA. Two distinctive characteristics of software products, complexity and invisibility, make the

development of QA methodology and its successful implementation a highly professional challenge [6].

2.3.6 QUALITY ASSURANCE

With the correctness-centered quality definitions adopted in **Section 2.3.3**, the crucial activities for QA are carried out to ensure that few or no defects remain in the software system when it is delivered to its customers or released to the market. Furthermore, QA activities also ensure that these remaining defects will cause minimal disruptions or damages.

Many QA alternatives are available to deal with software defects. A thorough examination of how various QA alternatives deal with software defects produces a generic classification scheme that can be utilized to assist developers better select, adapt and use different QA alternatives and related techniques for specific applications. In the next section, a classification scheme initially proposed in Tian [13] is described.

2.3.6.1 A CLASSIFICATION SCHEME

Different QA activities are viewed as attempting to prevent, eliminate, reduce, or contain various specific problems relating to product defects. Such QA alternatives can be classified into three generic categories, described below [9]:

1. *Defect prevention through error blocking or error source removal:* These blocking or removal QA activities lead to the prevention of certain types of faults from being injected into the software. It should be noted that errors are the missing or incorrect human actions that lead to occurrence of faults into software systems. These actions can directly be blocked or corrected, or the underlying causes for them can be removed. Therefore, defect prevention can be carried out using two generic ways:
   a) *Eliminating certain error sources,* by eliminating the root causes for the errors, for example by eliminating ambiguities or correcting human misconceptions,.
   b) *Fault prevention or blocking,* this is done by directly correcting or blocking these missing or incorrect human actions. Such techniques disconnect the

causal relation between error sources and faults by using certain tools and technologies, by enforcement of certain process and product standards, etc.

2. *Defect reduction through fault detection and removal:* These QA techniques first detect and then remove certain faults immediately after being injected into the software systems. It should be noted that most traditional QA activities belong to this category. For example,
   a) Inspection directly detects and removes faults from the software code, design, etc.
   b) Testing removes faults based on related failure observations during program execution.

   Various other QA techniques are applied to reduce the number of faults in a software system. Such techniques are based on either static analyses or observations of dynamic executions.

3. *Defect containment through failure prevention and containment:* These containment measures focus on software failures. They contain them to local areas and prevent them to be global failures observable to end users. They also limit the damage caused by software system failures. Therefore, defect containment are usually carried out in two generic ways:
   a) Some QA alternative techniques, such as the use of fault-tolerance techniques, act to break the causal relation between faults and failures in order to prevent them causing global failures, thus “tolerating” these local faults.
   b) A technique that extends fault-tolerance is applying the containment measures that act to avoid catastrophic consequences, such as death, personal injury, and severe property or environmental damages, in case of failures.

For the purpose of this research, different defect prevention techniques, organized in the above classification scheme will be surveyed next.

2.3.6.2 DEFECT PREVENTION

The QA alternatives commonly referred to as “defect prevention activities” can be used for most software systems in two ways: for reducing the chance for defect injections and for minimizing the subsequent cost to deal with these injected defects.

Most of the defect prevention activities assume that there are known error sources or missing/incorrect actions that result in fault injections, as follows [9]:

1. In cases where human misconceptions are the error sources, education and training should be used to remove these error sources.
2. In cases where imprecise designs and implementations deviate from product specifications or design intentions are the sources of faults, formal methods should be used to prevent such deviations.
3. In cases where non-conformance to selected processes or standards is the problem source that leads to fault injections, then process conformance or standard enforcement should be employed to prevent the injection of related faults.
4. In case certain tools or technologies are capable to reduce fault injections under similar environments, they should be adopted.

Based on what stated above, root cause analyses should be carried out to establish these preconditions, or *root causes,* for injected or potential faults. Knowing these error sources, will enable the application of appropriate defect prevention activities in order to prevent injection of similar faults in the future.

1. Education and Training

Education and training QA technique represents people-based solutions for error source elimination. As observed by many software practitioners, human factor is crucial factor in determining the quality and, most importantly, the success or failure of most software projects. Therefore, education and training of software professionals assist them in controlling, managing, and improving the way they work. Education and training activities can further help them to ensure that they have no or few misconceptions concerning the software product and its development. The elimination of these human misconceptions will eventually lead to the prevention of injecting various types of faults into software products. It should be stated that education and training effort meant for error source elimination should focus on the following areas [9]:

1. *Product and domain specific knowledge.* If the concerned people do not possess adequate knowledge about the product type or application domain, wrong solutions will likely be implemented.

2. *Software development knowledge and expertise* play a crucial role in the development of high-quality software products.
3. *Knowledge about development methodology, technology, and tools* also plays an important role in the development of high-quality software products.
4. *Development process knowledge.* If the project team does not have a good understanding of the product development process, that development process will not be implemented correctly.

2. Formal Methods

Formal development methods or formal methods in short represent another way to eliminate certain error sources and also a way to verify the absence of related faults. Such methods include formal specification and formal verification. The former is concerned with producing a clear set of product specifications that deal with customer requirements, as well as environmental constraints and design intentions, thereby reducing the chances of accidental fault injections. The latter checks the conformance of software design or code against these formal specifications in order to ensure that the produced software is fault-free regarding its formal specifications. Different techniques are available to specify and verify the correctness of software systems, these are [9]:

1. The earliest and most effective formal method is termed “axiomatic approach” [2] [14]. In this method, the meaning of a program element or the formal interpretation of the effect of its execution is abstracted into an axiom. The method uses more axioms and rules in order to connect different pieces together. *Preconditions* is a term denoting a set of formal conditions describing the program state before the execution of a program, while *postconditions* is a term used to denote the set of formal conditions after program execution. This approach is used to verify that a given program satisfies both its prescribed preconditions and postconditions.
2. Other influential formal verification techniques include the *predicate transformer* based on weakest precondition ideas [15] [16], and *program calculus* or *functional approach* heavily based on mathematical functions and symbolic executions [17]. The basic ideas of these techniques are similar to that of the axiomatic approach, however the proof procedures are somewhat different.

3. There also exist other limited-scope or semi-formal techniques. Such techniques check certain properties instead of verifying the full correctness of programs. An example of such methods are the model checking techniques which are gaining presently popularity in the software engineering research community [18]. It is noteworthy that semi-formal methods that are based on forms or tables [19] instead of formal logic or mathematical functions are also applied and gained importance.

So far, the biggest obstacle of using formal methods is the high cost associated with the difficult task of performing these human-intensive activities correctly without adequate automated support. This fact also explains, to a degree, the increasing popularity of limited scope and semi-formal approaches.

3. Other Defect Prevention Techniques

Other defect prevention techniques including those based on technologies, tools, processes, and standards, are briefly discussed below [9]:

1. Besides the formal methods surveyed above, appropriate use of other software methodologies or technologies can also help reduce the chances of fault injections. Problems of low quality “fat software” can be prevented by using disciplined methodologies and by resorting to essentials for high-quality “lean software” [20]. In the same way, using information hiding principle [21] will reduce the complexity of program interfaces and interactions among different components, and eventually reduce the occurrence possibility of related problems.
2. Processes should be managed in better ways in order to eliminate many systematic problems. For example to avoid inconsistencies or interface problems among different software components, a process should be better defined and followed for system configuration management. Better definition and conformance of a process ensures the elimination of many error sources. Another methodology used for reducing failure injection is the enforcement of selected standards for certain types of products and development activities.
3. In some cases, specific software tools are used to reduce the chances of fault injections. An example of this is that a syntax-directed editor that automatically

balances out each open parenthesis, "{", with a close parenthesis, "}", is frequently used to reduce syntactical problems in programs written in the C language.

It goes without saying that additional effort is required to direct the selection of suitable processes, standards, tools, and technologies, or to modify existing ones to suit specific application environments. Effective monitoring and enforcement systems are also required to make sure that the adopted processes or standards are followed, or that the appropriate technologies are used in a proper way, to reduce the possibility of fault injections.

SUMMARY

This chapter introduces the discipline of software engineering with all its specific characteristics and paradoxes, contrasts it with more traditional engineering disciplines, and elucidates the role that software testing plays within this discipline. It also defines software, software quality and software quality assurance and introduces different software quality attributes. It distinguishes between software errors, software faults and software failures and identifies the various causes of software errors. This chapter also describes different QA alternatives and offers a comprehensive coverage of defect prevention techniques for QA such as education and training, formal specification and verification, and other defect prevention techniques including those based on technologies, tools, processes, and standards.

CHAPTER THREE
SOFTWARE SPECIFICATION AND VALIDATION

3.1 SOFTWARE SPECIFICATIONS

In general, specification of a software product is defined as the description of the functional requirements that the product is expected to satisfy. Particularly IEEE standards define a specification as [5]:

"A document that specifies, in a complete, precise, verifiable manner, the requirements, design behavior, or other characteristics of a system or component, and, often, the procedure for determining whether these provisions have been satisfied."

Studying software specifications in the context of software verification and testing is not a very common practice; however, this is done here for a variety of reasons [7]:

1. *Testing in the Context of Verification and Validation*. The best way to view testing is as part of a broader policy of verification and validation. Moreover, studying software specifications provides an opportunity to exercise the activities of software verification and validation, and to identify the role that testing plays in these important activities.
2. *A Bridge between Testing and Verification*. Both dynamic testing and static verification, as commonly argued, are important complementary techniques to ensure the correctness or reliability of software products. However, for complementarity to be meaningful the results of these two techniques should be expressed in the same broad framework, this is possible through specifications.
3. *A Basis for Hybrid Verification*. In the recurrent or repeated debate concerning the comparative merits or virtues of "static analysis" and "dynamic testing", people often neglect an important detail: the observation that what makes a method ineffective is not any intrinsic shortcoming, but rather the fact that it is used with the wrong specification. Therefore, a cost effective approach to software quality may be to use testing for some parts of the specification, and use static analysis for other parts. However, this approach is applicable only when these two methods are devised to deal with the same specification framework.

4. *Specifications are the basis for test oracles*. As known, designing a test oracle is an important step in testing a software product; such a step consists mainly of selecting a specification against which the program is tested, and in implementing it. The importance of the step stems from the important role it plays in determining the effectiveness of the test.

Taking the above-mentioned facts into consideration, the topic of software specifications is discussed because it constitutes an important aspect of the study of software testing. A wide range of specification languages are used in research circles. However, some of them are relatively widely used in the industry.

3.2 A DISCIPLINE OF SPECIFICATION

The *specification* of a software product is generally defined as "a description of the properties that the product must have to fulfill its purpose" [7]. A software specification is usually derived by identifying all the concerned stakeholders of the (existing or planned) software product, eliciting the requirements that these stakeholders require the software product to fulfill, formulating and combining these requirements, and compiling them into a cohesive document [7]. Specifications typically relate to both functional and operational requirements. However, the focus is mainly on the functional ones in this research, i.e. those specifications dealing with the input/ output behavior of a software product.

3.2.1 SPECIFICATION: THE PROCESS AND THE PRODUCT

Literarily and scientifically (software engineering-related), the word specification refers both to a process and a product [22]. When meaning a product, the specification plays two essential roles: first, it acts as a contract that binds the user (it defines his/her requirements) and the designer (it defines his/her obligations); second, it acts as main working document for the designer [22]. When meaning a process, it consists of two steps [22]: the specification generation step, when the specification is progressively constructed from requirements data elicited from the user (group) by the specifier group; the specification validation step, when the specification is matched against redundant requirements data elicited from the user (group) by the verification and validation group.

3.2.2 THE SPECIFICATION PRODUCT: ITS THREE FORMS

According to Boudriga [22] there are three forms of specifications, which will be discussed below in turn:

1. *Procedural specifications*. These are specifications that describe a program that is intended to assign correct outputs to legal inputs. Various calls of the program with a specified input produce the same output. As examples, these include programs like a sort package program, a square root program, and a tree traversal program.
2. *Data type specifications*. These specifications describe the behaviour of a data type, whose state can be modified or probed by means of predefined procedures and functions. A call of a procedure or function produces a result that relies, by definition, on previous calls of procedures. Typical examples of such systems include abstract data types, databases etc.
3. *Continuous process specifications.* Such specifications characterize programs that carry out a stimulus-response mechanism. The programs' response to a particular stimulus depends not only on the current stimulus, but also on the previous stimuli submitted to the system. Such programs include operating systems, industrial process monitoring systems, and embedded systems.

3.2.3 PROPERTIES OF SPECIFICATIONS

For a specification to carry out its function in a sufficient way, it should satisfy numerous properties. In the past, many researchers have suggested a number of required properties that specifications must have. Some like Hence; Parnas [23] recommend these properties: completeness, minimality, formality and relevance. Others like Liskov and Zilles [24] favor the following properties: formality, constructability, comprehensibility, minimality, applicability and extensibility. Meyer [25] recommends what he termed "the seven sins of the specifier": noise, silence, over-specification, contradiction, ambiguity, forward reference and wishful thinking. Although they are apparently different (some are more colorful than others), they widely agree regarding desirable properties in specifications. Here, the description of Boudriga is used [22] who regard it very normal to distinguish between properties of the product and properties of the process, as stated below:

1. *Properties of the product.* The properties of the specification as a product are the ones that can be identified by examining the product itself, regardless of how it relates to the user's requirements. A specification of a product should satisfy two conditions, namely:
    1. *Formality*. The specification must be represented in such a way as to describe precisely what functional behavior is required (written in a language whose syntax and semantics are formally defined).
    2. *Abstraction*. The specification must describe what requirements the software product must satisfy, not how to satisfy them. In short, the specification is expected to focus on WHAT and not HOW and containing no design or structural details. HOW to satisfy requirements is the concern of the designer.

2. *Properties of the process.* These are concerned with the relationship between the desire of the user and the specifications. Being a process, a specification should satisfy two conditions [7]:
    1. *Completeness*. The specification should be complete by capturing *all* the relevant requirements of the product.
    2. *Minimality*. The specification should be minimal by capturing *nothing but* the relevant requirements of the product.

A specification that satisfies these two conditions (completeness and minimality) is called *valid*. In **Section 2.5**, how to ensure the validity of specifications will be discussed.

3.2.4 DATA TYPES

Regarding specifications, a data type is usually composed of a group of operations performed on a set of objects, with the constraint that the behaviour of the objects can only be observed during application of the operations [24]. One common example of data type is the stack. The class of objects comprises all possible stacks and the set of operations includes the traditional operations of push, pop, top, and operation init that creates new stacks. Boudriga [22] named three different kinds of operations. First, the *primitive constructors* are parameter-free operations that produce a result in the class defined by the data type; an example of such is operation *init*. The

combinatorial constructors map objects of a class (and eventually some additional parameters) into other objects; examples of such are operations push and pop of a stack data type. The *accessors* report on the data type but do not affect its subsequent behaviour; an example of such operation is the operation top of the stack data type. A number of ADTs have shown their usefulness in many different applications, these include: List, Queue, Stack, String, Tree, etc. Each of them can be defined in several various non-equivalent ways. For example, a stack ADT may have a count operation that shows how many items have been pushed and not yet popped. Such choice makes a difference both for its clients and its implementation.

3.3 FORMAL SPECIFICATION

To achieve progress, 'traditional' engineering disciplines like electrical or civil engineering, develop better mathematical techniques. The engineering industry also recognized the need for mathematical analysis and incorporated mathematical analysis into its processes [26].

Mathematical analysis became a regular part of the process of development and validation of a product design. However, software engineering did not rely on mathematical analysis. There was of course research conducted over the years to adopt the use of mathematical techniques in the software process. However, these techniques were discovered to be of a limited impact. Formal methods of software development are not largely used in the industry of software development. One reason is that most software development companies do not apply these methods in their software development processes because they find them to be not cost-effective. 'Formal methods', as a term, is used to mean activities that depend on mathematical representations of software. These include activities that rely on formal specification such as formal system specification, specification analysis and proof, transformational development, and program verification [26].

A proposal developed by numerous software engineering researchers in 1980s claimed that the application of the formal development methods was the best way to improve software quality. They state that formal methods include rigor and detailed analysis that would enable the development of programs with fewer errors and suitable for users' needs. They went on predicting that, by the 21st century, a large proportion of

software would be developed using formal methods. Evidently, their prediction has not come true. Four major reasons made this prediction untrue [26]:

1. *Successful software engineering.* Alternative software engineering methods such as structured methods, configuration management and information hiding were used in software design and development processes resulting in better software quality. This application of other methods proved that formal methods are not the only way to improve software quality.
2. *Market changes.* During 1980s, software quality was looked at as the main software engineering problem. However, by the end of that decade, the key issue for several software development organizations is not quality but time-to-market. Software development should be fast, since customers may in most cases accept software with some faults if quick delivery is accomplished. However, techniques for fast software development do not apply effectively with formal specifications. Of course, software quality still matters but it must be combined with rapid delivery to be more valuable.
3. *Limited scope of formal methods.* Formal methods do not contribute very much in specifying user interfaces and user interaction. The user interface component constitutes an important part of most systems; therefore, formal methods are only used in the development of the other parts of the system.
4. *Limited scalability of formal methods.* Formal methods still do not scale up well. Therefore, formal methods techniques have in most cases been applied to develop relatively small, critical kernel systems. Systems with larger size, require much time and effort to be developed using formal methods.

The above-mentioned factors indicate that most software development companies do not intend to apply formal methods in the development process. Alternatively, formal specification is a better method of detecting specification errors and presenting the system specification in clear way. However, organizations that invested in applying formal methods ended up with fewer errors in software they delivered and at the same time without an increase in development costs. This shows that formal methods can be cost-effective if application is limited to key parts of the system and if companies are intending to make the high initial investment in this technology [26].

However, formal methods is still increasingly used in developing critical systems, in which emergent system properties including safety, reliability and security are very important. Failure in these systems results in high cost, which means that companies are ready to bear the high introductory costs of formal methods to make sure that the developed software is as dependable as possible. It is known that critical systems have very high and increasing validation and failure costs. Such costs can be reduced by formal methods [26].

3.3.1 CLASSIFICATION OF FORMAL SPECIFICATION METHODS

Formal specification methods use languages with mathematically defined syntax and semantics, and offer methods to describe systems and their properties. The power of formal methods lies on the level of formality and expressiveness provided by their specification languages. Their strength also lie in the availability of tools that enable them to develop the system in a way that strictly satisfies the set specifications of the system. Thus, a formal method could be categorized depending upon its strength and utilization, which in turn rely on the four pillars mathematical basis, type of systems, level of formality, and tools support [27].

In a formal specification language, the mathematical basis is either one of these concepts: algebra, logic, set theory, and relations or some combination of them. To get the desired benefits of a formal method, its specification language should provide formal extensional features. These features adequately address software engineering concerns and allow specifications to be composable from simple structured units, to be generic and parameterized, and to have well-described interfaces. Generally, systems can be broadly classified into three categories (with or without time dimension): sequential, concurrent, and distributed. In most cases, a specification language is designed to satisfy the need of systems classified under one of these categories in this broad classification. A reactive system may involve time constraints and exhibit concurrency. A transaction system, such as database systems and web-based service-oriented systems, is usually distributed and involves timing constraints. Systems such as telephone switching systems, control systems, and transportation systems involve both sequential and concurrent behaviour and could be governed by strict timing constraints. Because of their complex behaviour, they require languages that support

constructs for specifying concurrent and time constrained interactions. It is common to combine two or more specification languages to specify these systems [27].

A formal language is mathematically based. However, it may or may not include a reasoning system to verify the system properties. But it is necessary to use a form of logic when reasoning about system behaviour. The verification method provided by a specification language should be closely tied to its mathematical basis.

There are a few exceptions, such as PVS (Prototype Verification System) [28], which offer both development and verification support.

For integrating formal method with software development, it is necessary to utilize tools support. Each specification must be correctly typed, checked for syntactic correctness, and semantically analyzed. Then, a specification that is syntactically and semantically error-free should also be subjected to refinement and verification. The aim of many specification languages is to carry out formal verification. They do not help with system development. However, many other specification languages provide development tools, but do not provide verification tools. There exist few exceptions which offer both development and verification support such as PVS [28].

Formal specification languages are broadly categorized as property-oriented, and model-oriented [22], which includes the state-machine-oriented case. A property-oriented approach constructs a theory and state everything that needs to be included in the theory. A model-based approach uses mathematical objects to build the model, and hence the properties of those objects can be freely used in the model.

3.3.1.1 PROPERTY-ORIENTED SPECIFICATION METHODS

Property-oriented category can be subdivided into two sub-groups namely; axiomatic and algebraic.

1. *First the axiomatic approach*: here, the objects are constructed from types, and operations on types are given as assertions in first-order predicate logic. This means that the axiomatic formal specification defines the semantics of functions of objects by describing the relations between different objects and functions as axioms (predicate-logical formula). For instance, the language Anna [29] uses assertions to annotate Ada programs. These assertions serve to verify the correctness of Ada programs. TAM'97 (Trace Assertion Method) [30] is an example of a formal method based on assertions for abstract specification of

module interfaces. Axiomatic approaches naturally lend to verification based on theorem proving method.

2. *Second the algebraic approach*: here, theory of objects and processes are defined as algebras. This method emphasizes the representation-free specification of objects and a declarative style of specifying their properties. An object is usually represented by a set of definitions, and an operation in an object is defined by a set of equations. Language terms are terms used in the underlying algebra and are not interpreted. Equations are changed into rewrite rules for generating new terms in the specification. In order for a property to be verified, it should be stated as a term. When a property is stated as a term in the algebra defined by the specification, it holds for the specified system. This means that algebraic specification methods provide both a language and a proof method. The best known algebraic specification language is OBJ3 [31]. OBJ algebraic specification languages family provides tool support for developing algebraic specifications and verifying stated properties. Specifically, OBJ3 affords modularity, genericity through parameterization, and extendibility through theory composition.

3.3.1.2 MODEL-BASED SPECIFICATION TECHNIQUES

It is known that every formal specification language constructs a model of the specified system. The significance of model-based languages comes from the fact that the model of the system is constructed in terms of mathematical objects such a sets, sequences and relations. These mathematical objects provide an abstract description of the state of the system, together with a number of operations over that state. In addition, these objects, particularly natural numbers, sets and relations, are utilized to construct a state model. An operation defined in a state is a function that modifies one or more state variables. Operation definitions use the underlying mathematical theory. The most widely known examples of model-oriented specification languages include Z [32], the B Method [33], and VDM [34].

3.4 THE SPECIFICATION OF DATA TYPES

Mili and Tchier [7] proposed a generalized, relation-based, model for the specification of software systems which comprises three types of specifications: the

procedural specifications (used for simple input output programs) as well as data type specifications (used for programs whose response depend not only on their input, but also on their internal state) and continuous process specifications. In this research, Mili's generalized model is used to the specification of data types.

3.4.1 A RELATIONAL MODEL

Recalling the discussion stated in **Section 3.2**, specifications should have two key attributes: formality and abstraction. Formality can be achieved by using a mathematical notation, which associates precise semantics to each statement. As for abstraction, it can be achieved by ensuring that the specifications describe the externally observable attributes of candidate software products, but do not specify, dictate or otherwise favor any specific design or implementation.

The following description of a stack data type is considered (One can find a detailed description of a stack and other Abstract Data Type such as Queue, Sequence, Set, Multiset and List in APPENDIX-A and https://sites.google.com/site/nahidahmedali/):

A stack is a data type that is used to store items (through operation push ()) and to remove them in reverse order (through operation pop()); operation top() returns the most recently stored item that has not been removed, operation size() returns the number of items stored and operation empty() tells whether the stack has any items stored; operation init() reinitializes the stack to an initial situation. If someone wants to stack, without saying anything about how to implement it. How would he or she do it?

Most data structure courses implement stacks by exhibiting a data structure composed of an array and an index into the array, and by explaining how push and pop operations affect the data structure. However, such an approach does not comply with the principle of abstraction since it specifies the stack by describing a possible implementation thereof. An alternative approach is to specify the stack by means of an abstract list, along with a list of operations, without specifying how the list is implemented. However, this also does not comply with the principle of abstraction, as it dictates a preferred implementation. In fact, a stack does not necessarily require a list of elements, irrespective of how the list is represented.

Hence, in order not to violate the abstraction principle, a model proposed by Mili and Tchier [7] is selected to be used to specify the stack. Such specification is done by

describing its externally observable behavior, without making any vague assumption about its internal structure. To this effect, a stack is specified by means of three parameters, as follows:

1. *An input space*, say X, which includes all the operations that may be invoked on the stack. Hence,

$$X = \{init, pop, top, size, empty\} \cup \{push\} \times itemtype,$$

where *itemtype* is the data type of the items we envision to store in the stack. In set X, inputs that affect the state of the stack (namely: OX= {init, push, pop}) and inputs that merely report on it (namely: VX= {top, size, empty}) should be distinguished.

From the set of inputs X, we build the set of input histories, H, where an input history is a sequence of inputs; this is needed because the behavior of the stack is not determined solely by the current input but involves past inputs as well.

- A set of input histories, $H = X^*$.

2. *An output space*, say Y, which includes all the values returned by all the inputs of VX. In the case of the stack, the output space is:

$$Y = (itemtype \cup \{error\}) \cup integer \cup Boolean,$$

Which correspond, respectively, to inputs top, size, and empty.

3. *A relation from H to Y*, which represents the pairs of the form (*h, y*), where h is an input history and *y* is an output that the specifier considers correct for h. This relation is denoted by stack, and the notation stack (*h*) is used to refer to the image of *h* by stack (if that image is unique) or to the set of images of *h* by stack (if *h* has more than one image). Below some pairs of the form (h, y) for relation stack:

1. stack(pop.init.top.pop.push(a).size.push(b).top.pop.push(c).top.pop.size.top)=**a**

2. stack(pop.push(a).pop.init.pop.push(a).size.top.pop.push(b).pop.top)=**error**.

3. stack(push(a).init.init.pop.push(b).top.size.push(c).push(d).top.pop.push(e).size)=**3**

4. stack(pop.init.pop.pop.push(a).empty.push(b).pop.top.push(c).push(d).empty)= **false**.

Description of possible input histories and corresponding outputs can go on this manner. How operations interact with each other is specified, but how each operation behaves is not prescribed. This leaves maximum latitude to the designer, as mandated by the principle of abstraction. It is clearly impractical to specify data types by listing

elements of their relations. In the next section, a closed form representation for such relations is explored.

3.4.2 AXIOMATIC REPRESENTATION

Mili and Tchier [7] represent the relation of a specification by means of an inductive notation. Induction on the structure of the input history is done. This notation includes two parts, namely:

1. *Axioms*, which represent the behavior of the system for trivial input histories.
2. *Rules*, which represent the behaviour of the system for complex input histories as a function of its behaviour for simpler input histories.

3.4.2.1 SPECIFICATION OF A STACK

As an illustration, the specification of the stack is represented by using axioms and rules. Throughout this presentation, let *a* be an arbitrary element of itemtype, and *y* an arbitrary element of *Y*; also, let *h*, *h'*, *h''* be arbitrary elements of *H*, and *h+* an arbitrary non-null element of *H*.

1. ***Axioms***. Axioms are used to represent the output of input histories that end with an operation in set *VX* (that reports on the state), namely in this case *top*, *size* and *empty*. It is understood that input histories that end with an operation in set *OX* (that affects the state) produce no meaningful output; hence, it is assumed that for such input histories, the output is any element of *Y*.
   - **Top Axioms**
     - stack(init.top) = error.
       Seeking the top of an empty stack returns an error.
     - stack(init.h.push(a).top) = a.
       Operation top returns the most recently stacked item.
   - **Size Axiom**
     - stack(init.size) = 0.
       The size of an empty stack is zero.
   - **Empty Axioms**
     - stack(init.empty) = true.
       A stack that doesn't contains any element is empty.

- o stack(init.push(a).empty) = false.

  A stack that contains element *a* is not empty.

2. ***Rules***. Whereas axioms characterize the behavior of the stack for simple input sequences, rules establish relations between the behavior of the stack for complex input histories and their behavior for simpler input histories.

- **Init Rule**
  - o stack(h.init.h') = stack(init.h').

    Operation init reinitializes the stack; whether sequence *h* intervened prior to init or did not, makes no difference for the future behavior (*h'*) of the loop.
- **Init Pop Rule**
  - o stack(init.pop.h) = stack(init.h).

    A pop operation on an empty stack has no effect: whether it occurred or did not occur makes no difference for the future behavior (*h*) of the loop.
- **Push Pop Rule**
  - o stack(init.h.push(a).pop.h+) = stack(init.h.h+).

    A pop operation cancels the most recent push: whether it occurred or did not makes no difference to the future behavior of the stack, though not to the present (if *h* ends with an operation in VX, we could not say that stack(init.h)=stack(init.h.push(a).pop) as the left hand side returns a specific value but the right hand side returns an arbitrary value).
- **Size Rule**
  - o stack(init.h.push(a).size) = 1+stack(init.h.size).

    Each push operation necessarily increases the size of the stack by 1, because the stack size is not bounded.
- **Empty Rules**
  - o stack(init.h.push(a).h'.empty) ⇒ stack(init.h.h'.empty)

    If, despite having operation push(a) in its history, the stack is empty, then a fortiori it would empty without push(a).
  - o stack(init.h.empty) ⇒ stack(init.h.pop.empty)

    If the stack is empty, then a fortiori it would be empty if an extra pop operation was performed in its past history.
- **VX-Operation Rules**

- stack(init.h.top.h+) = stack(init.h.h+).
- stack(init.h.size.h+) = stack(init.h.h+).
- stack(init.h.empty.h+) = stack(init.h.h+).

  VX operations have no impact on the future behaviour of the stack, by definition, since all they do is to enquire about its state.

A closed form specification of the stack has been written, in such a way that the externally observable properties of the stack are described closely, without any reference to how a stack ought to be implemented; a programmer who reviews this specification has all the latitude he/ she needs to implement this stack as he/ she sees fit.

3.4.2.2 SPECIFICATION OF A QUEUE

How to represent the specification of a queue is discussed, in the same way that the specification of a stack is written above. The input space (from which we infer the set of input histories) is represented in turn, then the output space, then the relation, which is denoted by queue.

1. *An Input Space*. let X be defined as:

$$X = \{\text{init, dequeue, front, rear, size, empty}\} \cup \{\text{enqueue}\} \times \text{itemtype}.$$

This set is portioned into O*X* = {init, enqueue, dequeue} and *VX* = {front, rear, size, empty}. let *H* be the set of sequences of elements of *X*.

2. *An Output Space*. let the output space be defined as:

$$Y = (\text{itemtype} \cup \{\text{error}\}) \cup \text{integer} \cup \text{Boolean}.$$

1. *Axioms*. The following axioms are proposed:

- **Front Axioms**
  - queue(init.front) = error.

    Invoking front on an empty queue returns an error.
  - queue(init.enqueue(a).enqueue(_)*.front) = a,

    Where enqueue(_)* designates an arbitrary number (including zero) of enqueue operations. Interpretation: Invoking front on a non empty queue returns the first element enqueued.
- **Rear Axioms**
  - queue(init.rear) = error.

Invoking rear on an empty queue returns an error.

  - queue(init.enqueue(_)*.enqueue(a).rear) = a,

    Invoking rear on a non empty queue returns the last element enqueued.

- **Size Axiom**
  - queue(init.size) = 0.

    The size of an empty queue is zero.

- **Empty Axioms**
  - queue(init.empty) = true.

    An initial queue is empty.
  - queue(init.enqueue(a).empty) = false.

    A queue in which an element has been enqueued is not empty.

2. ***Rules***. The following rules are proposed.

- **Init Rule**
  - queue(h.init.h') = queue(init.h').

    The init operation reinitializes the queue, i.e. renders all past input history irrelevant.

- **Init Dequeue Rule**
  - queue(init.dequeue.h) = queue(init.h).

    A dequeue operation executed on an empty queue has no effect.

- **Enqueue Dequeue Rule**
  - queue(init.enqueue(a).enqueue(_)*.dequeue.h+)=

    queue(init.enqueue(_)*.h+)

    A dequeue operation cancels the first element enqueued, by virtue of the FIFO policy of queues.

- **Size Rule**
  - queue (init.h.enqueue(a). size) = 1+ queue(init.h.size).

    An enqueue operation increases the size of the queue by 1.

- **Empty Rules**
  - queue(init.h.enqueue(a).h'.empty) ⇒ queue(init.h.h'.empty)
  - queue(init.h.empty) ⇒ queue(init.h.dequeue.empty)

    Removing an enqueue or adding a dequeue to the input history of a queue makes it more empty.

- **VX-Operation Rules**
  - queue(init.h.front.h+) = queue(init.h.h+).
  - queue(init.h.rear.h+) = queue(init.h.h+).
  - queue (init.h.size.h+) = queue(init.h.h+).
  - queue(init.h.empty.h+) = queue(init.h.h+).

    VX operations leave no trace of their passage; once they are serviced and another follows them, they are forgotten: whether they occurred or did not occur has no impact on the future behavior of the queue.

Other ADT's (namely: Sequence, Set, Multiset, List) specification is presented in APPENDIX-B and https://sites.google.com/site/nahidahmedali/.

3.5 SPECIFICATION VALIDATION

The software engineering literature contains examples of software projects that fail. The reason for such failures is not programmer's inability of how to write code or how to test it. The reason is that analysts and engineers fail to write valid specifications that capture all the relevant requirements (for the sake of completeness) and nothing but relevant requirements (for the sake of minimality) [7]. As a result, it is crucial to validate specifications for both completeness and minimality, and to invest the necessary resources to this effect before proceeding with subsequent phases of the software lifecycle. In this section, the process of specification validation is briefly and cursorily discussed. It is discussed in the narrow context of the relational specifications that were introduced in the previous section. The modest goal is to give the reader some sense of what it may mean to validate a specification.

In the previous section, the specifications on the basis of the proposed requirement are written and the completeness and minimality of candidate specifications can be judged by considering the same source, i.e. the proposed requirement. Tasking the same person or group with generating the candidate specifications and judging their validity (completeness and minimality) will lead to the result that the same biases that cause the person or group to write invalid specifications may cause him/ her to overlook the invalidity of their specification. The only way to make sure that a measure of confidence is not bias in the validation of the specification is to employ separate teams. One for the generation of the specifications and one for the validation of those

specifications. With this respect, Mili and Tchier [7] devise the following two-team, two-phase approach as shown in Table 3.1.

Table 3.1: Two-Team, Two-Phase Approach

Activity / Phase	Specification Generation	Specification Validation
Specification Generation	Generating the specification from sources of requirements.	Generating validation data from the same sources of requirements.
Specification Validation	Updating the specification according to feedback from the validation team.	Testing the specification against the validation data generated above.

These two phases are described below:

1. *The Specification Generation Phase*. In this phase, the specification team generates the specification by referring to all the sources of requirements (requirements documents). By referring to the same requirement documents or sources, the validation team generates validation data that it intends to test the specification against. Mili and Tchier [7] distinguished between two types of validation data:
    1. *Completeness Properties*. These are properties that the specification must have, but the validation team suspects the specification team may fail to record.
    2. *Minimality Properties*. These are properties that the specification must not have, but the validation team suspects the specification team may record them inadvertently.

    For the sake of redundancy, the specification team and the validation team must work independently of each other.
2. *The Specification Validation Phase*. In the specification validation phase, the validation team tests the specification against completeness and minimality data generated in the previous phase, while the specification team updates the specification, if it turns out that it was not complete or not minimal.

It remains to discuss: what form does the validation data take, and how does one test a specification against the generated validation data. The answers to these questions are given in the following section where abstract data type specification validation is discussed.

3.5.1 ADT'S SPECIFICATION VALIDATION

In the previous section, specifications of a number of ADT's have been written, namely: a stack and a queue. How does one know that such specifications are valid? Valid means that they capture all the properties one wants them to capture (completeness) and nothing else (minimality). To bring a measure of confidence in the validity of these specifications, a validation process is envisioned in which the focus will solely be on completeness for the sake of simplicity. One imagines that while these specifications are written, an independent verification and validation team is generating formulas of the form stack(h) = y for different values of h and y, on the grounds that whatever one writes in his/her specification should logically imply these statements. Then the validation step consists of checking that the proposed formulas can be inferred from the axioms and rules of the specification. If they do, then one can conclude that such specification is complete with respect to the proposed formulas; if not, then one needs to check with the verification and validation team to see whether his/her specification is incomplete or perhaps the validation data is erroneous [7].
For the sake of illustration, one checks whether his/her specification is valid with respect to the formulas written in **Section 3.4.1** as sample input /output pairs of stack specification (One can find other ADT's validation data in APPENDIX-C and https://sites.google.com/site/nahidahmedali/):

1. stack(pop.init.top.pop.push(a).size.push(b).top.pop.push(c).top.pop.size.top)=**a**
2. stack(pop.push(a).pop.init.pop.push(a).size.top.pop.push(b).pop.top)=**error**.
3. stack(push(a).init.init.pop.push(b).top.size.push(c).push(d).top.pop.push(e).size)=**3**
4. stack(pop.init.pop.pop.push(a).empty.push(b).pop.top.push(c).push(d).empty)= **false**.

He/she finds:

1. stack(pop.init.top.pop.push(a).size.push(b).top.pop.push(c).top.pop.size.top)=**a**
   = {by virtue of the init rule}

stack(init.top.pop.push(a).size.push(b).top.pop.push(c).top.pop.size.top)

= {by virtue of the VX-op rules}

stack(init.pop.push(a).push(b).pop.push(c).pop.top)

= {by virtue of the push-pop rule, applied twice}

stack(init.pop.push(a).top)

= {by virtue of the second top axiom, with h = < pop >}

a. QED

2. stack(pop.push(a).pop.init.pop.push(a).size.top.pop.push(b).pop.top)=**error**.

= {by virtue of the init rule}

stack(init.pop.push(a).size.top.pop.push(b).pop.top)

= {by virtue of the init-pop rule}

stack(init.push(a).size.top.pop.push(b).pop.top)

= {by virtue of the VX-op rules}

stack(init.push(a).pop.push(b).pop.top)

= {by virtue of the push-pop rule, applied twice}

stack(init.top)

= {by virtue of the first top axiom}

error. QED

3. stack(push(a).init.init.pop.push(b).top.size.push(c).push(d).top.pop.push(e).size)=**3**

= {by virtue of the init rule}

stack(init.pop.push(b).top.size.push(c).push(d).top.pop.push(e).size)

= {by virtue of the VX-op rules}

stack(init.pop.push(b).push(c).push(d).pop.push(e).size)

= {by virtue of the push-pop rule}

stack(init.pop.push(b).push(c).push(e).size)

= {by virtue of the init-pop rule}

stack(init.push(b).push(c).push(e).size)

= {by virtue of the size rule, with h = <push(b).push(c)>}

1 + stack(init.push(b).push(c).size)

= {by virtue of the size rule, with h = <push(b)>}

1 + 1 + stack(init.push(b).size)

= {by virtue of the size rule, with h = < >}

1 + 1 + 1 + stack(init.size)

= {by virtue of size axiom}

1 + 1 + 1 + 0

= {arithmetic}

3. QED

4. stack(pop.init.pop.pop.push(a).empty.push(b).pop.top.push(c).push(d).empty)= **false**.

= {by virtue of the init rule}

stack(init.pop.pop.push(a).empty.push(b).pop.top.push(c).push(d).empty)

= {by virtue of the init-pop rule, applied twice}

stack(init.push(a).empty.push(b).pop.top.push(c).push(d).empty)

= {by virtue of the VX-op rules}

stack(init.push(a).push(b).pop.push(c).push(d).empty)

= {by virtue of the push-pop rule}

stack(init.push(a).push(c).push(d).empty)

=>{by virtue of the first empty rule, with h=<push(a)>, h'=< push(d)>}

stack(init.push(a).push(d).empty)

=>{by virtue of the empty rule, with h=<push(a)>, h'=< >}

stack(init.push(a).empty)

=>{by virtue of the second empty axiom}

false. QED

Because the specification given has survived four tests unscathed, we gain a bit more confidence in its validity.

SUMMARY

This chapter introduces the concept of software specifications as both a process and a product. It discusses the three forms of specification product and the properties of the specifications. In addition, the chapter discusses the formal specification and its role in improving software quality. It also, discusses the classification of formal specification methods into: property-oriented specification methods and model-based specification techniques. Finally, it describes the relational specification and the axiomatic representation of the relational specification of

systems that maintain an internal state. It concludes with the generation and validation of axiomatic specifications of abstract data types.

CHAPTER FOUR
PROGRAM CORRECTNESS AND VERIFICATION

4.1 INTRODUCTION

Today’s software systems have become essential parts of human everyday life. They are currently utilized in almost all fields of human activities. More and more of them are required, and their complexity increases continuously. Software critical systems require error-free programming [35]. To obtain error-free programs, a new way of writing programs and much better educated programmers are needed.

There exists a tradeoff between the desire to obtain a system as quickly as possible, and to build a correct system. The experiences show that numerous finished software products have errors during maintenance, and in developing a software product much effort will be spent on testing and debugging [35]. Nevertheless, some errors are not detectable by testing, and some of them will never be detected. Due to unreliable software, many people died, and serious economic damages have been experienced [35].

Program verification is a technique used to ensure or verify the correctness of software products, it’s a static method that attempts to prove by logical reasoning that the program is correct, assuming a given semantic definition of the source language. A complementary technique is *software testing*, which is the activity where a software product is executed on sample data and its behavior is judged with respect to the specification that the program is intended to satisfy. These two approaches to achieve program quality assurance are usually considered as alternatives, and offer contrasting attributes [36]:

1. Whereas *program verification* is a static technique, which operates on the source code, *program testing* is a dynamic technique that uses the executable code of the program.
2. Whereas *program verification* requires the assumption that the compiler and the run time environment of the program are compatible with the semantic definition of the programming language; *program testing* requires the assumption that the testing environment is a faithful imitation of the user-operating environment.

3. Whereas *program verification* can be used to prove the absence of faults (under the assumptions cited above) but cannot necessarily be used to prove their presence (if a proof fails, there is no easy way to tell whether it is because the program is incorrect or because the proof was not well planned); *program testing* can be used to find faults (if a test fails) but cannot be used to prove the absence of faults (it is virtually impossible in general to test the program on all possible test data)
4. Whereas *program testing* can be applied to programs of any size and complexity, *program verification* can only be applied to very small programs, which use only simple constructs.
5. Whereas *program verification* concludes with a logical claim about the correctness of the program, *program testing* concludes either with a report of a failure or some statistical claim about the reliability of the program.

In practice, program verification has been the subject of much research, but has made little inroads into industrial practice; by contrast, program testing has been common practice in the industry, but in circumstances where it is difficult to make any credible claims of software quality. However, program correctness is an important concept in program testing for several reasons, some of which are discussed below [7]:

1. The focus of software testing is to run the candidate program on selected input data and check whether the program behaves correctly with respect to its specification. The behavior of the program can be analyzed only if one knows what a correct behavior is. Hence, the study of correctness is an integral part of software testing.
2. The study of program correctness leads to analyzing candidate programs at arbitrary levels of granularity. In particular, it leads to make assumptions on the behavior of the program at specific stages in its execution, and to verify (or disprove) these assumptions. The same assumptions can be checked at run-time during testing, producing valuable information as one tries to diagnose the program or establish its correctness. Hence, the skills that one develops as he/she tries to prove program correctness enable him/her to be testers that are more effective.

3. It is common for program testers and program provers to make polite statements about testing and proving being complementary, then to assiduously ignore each other (each other's method). However, there is more to complementary than meets the eye. Very often, an intrinsic attribute of a testing method or a proving method does not make it ineffective, but rather the fact that the method is used against the wrong type of specification. Hence, it is advantageous, given a complex/compound specification, to decompose it into two broad components: one that lends itself to testing and one that lends itself to proving; and apply each method against the appropriate specification component. Hence, by doing this one achieves great gains in efficiency, quality, and reliability.
4. It is best to view software testing, not as an isolated effort, but rather as an integral part of a broad, multi-pronged policy of quality assurance, that deploys each method where it is most effective, by virtue of the *Law of Diminishing Returns*.

The present chapter introduces the concepts of program correctness, program verification, Hoare Logic, and formal verification techniques and gives some examples of proving programs correctness.

4.2 CORRECTNESS OF PROGRAMS

A *correct program* is one that does exactly what its designers and users intend it to do– doing more does not preclude correctness [37]. A *formally correct program* is one whose correctness can be proved mathematically, at least to a point that designers and users are convinced about its relative absence of errors [37]. For a program to be formally correct, there must be a way to specify precisely (mathematically) what the program is intended to do, for all possible values of its input, these so-called specification languages are based on mathematical logic [37]. The proof of correctness is defined by IEEE as [5]:

1. "A formal technique used to prove mathematically that a computer program satisfies its specified requirements."
2. "A proof that results from applying the technique in (1)."

The concept of program correctness was introduced by Floyd [38]. In addition, a method for proving program correctness was presented in the same paper. Before, the

correctness of individual algorithms was proved by Hoare [2], [39], Foley and Hoare [40], London [41], [42], Naur [43].

The ideas of Floyd were developed by Hoare [2] who introduced an axiomatic method for proving the partial correctness of a program (introduced in **Section 4.3.1** of this chapter.). Then Dijkstra [15] introduces the important concept of weakest precondition. His work that formally deriving correct programs from specifications, was continued by Gries [44] who states that it is more important to develop correct programs than to prove later their correctness. A program and its proof should be developed hand-in-hand with the former usually leading the way. This idea was developed further by Dromey [45], and Morgan [46].

Some people [47] are against proving correctness. Others consider expensive, since they think that the effort to build a program in such a way is considerably increased. Certainly, proving correctness of a real large program is too complicated and, maybe, error-prone, which defeats the purpose of the proof effort.. However, the correctness of the important and critical procedures used in the system may be proved. In addition, the client does not frequently change the specifications of these procedures, as some researchers argue against proving correctness. Moreover, proving correctness has an important impact on program verification. It is well known that proving correctness and testing complete each other [35].

Formally proving the correctness of a small program, of course, does not address the major problem facing software designers today. Modern software systems have millions of lines of code, representing thousands of semantic states and state transitions. This innate complexity requires that designers use robust tools for assuring that the system behaves properly in each of its states [37].

In fact, a complete program correctness proof consists of two parts: a partial correctness proof and a termination proof. A partial correctness proof shows that a program is correct when indeed the program halts (or terminate). However, a partial correctness proof does not establish that the program must halt. To prove a program always halt, the proof is called "T*ermination Proof*" [48]. IEEE defined partial correctness and total correctness as [5]:

1. “Partial correctness. In proof of correctness, a designation indicating that a program’s output assertions follow logically from its input assertions and processing steps.”
2. “Total correctness. In proof of correctness, a designation indicating that a program’s output assertions follow logically from its input assertions and processing steps, and that, in addition, the program terminates under all specified input conditions.”

4.3 SOFTWARE VERIFICATION

The semantics of programs are necessary to introduction for making explicit exactly what programs do. The central issue is how to establish that a program does the right things. The beginning will be with that the program is syntactically correct. The detection of syntax errors (e.g. missing parentheses and the like) belongs to the study of formal programming languages. It is important to detect semantic errors, incorrect loop initialization etc. [49]. A common approach to the problem of program correctness is the program verification. Program verification will be the subject matter of the rest of this chapter. IEEE defines program verification as [5]:

1. “The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.”
2. “Formal proof of program correctness.”

This approach mathematically proves that a program satisfies its specifications. The definition of these specifications must incorporate mathematical precision. In the case where a formal definition of these specifications is available (for instance, as a formula of predicate logic) the program verification may be performed formally, even with the aid of a computer. A major difficulty when actually applying the techniques of program verification to real-life programs is the combinatorial complexity of large proofs. One way to overcome this difficulty is by proposing new program verification system [49]. This research provides a systematic exposition of one of the most common approaches to program verification, namely program verification using Hoare Logic.

4.3.1 HOARE LOGIC

Hoare logic (also known as Floyd–Hoare logic or Hoare rules) is a formal system with a set of logical rules for reasoning rigorously about the correctness of computer programs. It was proposed in 1969 by the British computer scientist and logician C. A. R. Hoare, and subsequently refined by Hoare and other researchers [2]. However, the original ideas were seeded by the work of Robert Floyd, who had published a similar system [38] for flowcharts.

4.3.1.1 HOARE TRIPLE

The central feature of Hoare logic is the Hoare triple, which describes how the execution of a piece of code changes the state of the computation. A Hoare triple is of the form:

$$\{p\}\, S\, \{q\} \tag{1}$$

Where S is a programming language statement or command, p and q are *assertions* (variables of) on the space of the program. Such Formulas are called Hoare formulas. p denotes the *precondition* and q denotes the *postcondition*: when the precondition is met, the command establishes the postcondition. Assertions are formulas in predicate logic.

Standard Hoare Logic proves only partial correctness, while termination needs to be proved separately. Thus the intuitive reading of a Hoare triple is:

Whenever p *holds* of the state before the execution of S, then q will hold afterwards, or S does not terminate. Note that if S does not terminate, then there is no "after", so q can be any statement at all. Actually, one can choose q to be false to express that S does not terminate. Total correctness can be also proven with an extended version of the while rule.

It is important to note that, Hoare logic provides axioms and inference rules for all the constructs of a simple imperative programming language.

4.3.1.2 AN INFERENCE SYSTEM

An inference system is a system for inferring conclusions from hypotheses in a systematic manner. Such a system can be defined by means of the following artifacts:

- A set *F* of (syntactically defined) formulas.
- A subset *A* of *F*, called the set of axioms, which includes formulas that one assumes to be valid by hypothesis.
- A set of inference rules, denoted by *R*, where each rule consists of a set of formulas called the premises of the rule, and a formula called the conclusion of the rule. One interprets a rule to mean that whenever the premises of a rule are valid, so is its conclusion. Usually, the rule is represented by listing its premises above a line and its conclusion below the line.

An inference in an inference system is an ordered sequence of formulas, say $v_1, v_2, ...v_n$. Each formula in the sequence, say v_i, is either an axiom or the conclusion of a rule whose premises appear prior to v_i, i.e. amongst $v_1, v_2, ...v_{i-1}$. A theorem of a deductive system is any formula that appears in an inference.

This section introduces an inference system that is used to establish the validity of Hoare formulas by induction on the complexity of the program component of the formulas [7]. To this effect, one presents in turn, the formulas, then the axioms, and finally the rules.

- **Formulas**. Formulas of the inference system include all the formulas of logic, as well as Hoare formulas.
- **Axioms**. Axioms of the inference system include all the tautologies of logic, as well as the following formulas:
  - $\{false\}\ S\ \{q\}$, for any program S and any postcondition q.
  - $\{p\}\ S\ \{true\}$, for any program S and any precondition p.
- **Rules**. One presents below a rule for each statement of a simple C-like programming language, in addition to a consequence rule that allows him/her to generalize a pre/post specification.
  1. Assignment Statement Rule: One considers an assignment statement that affects a program variable (and implicitly preserving all other variables), and interprets it as an assignment to the whole program state (changing the selected variable and preserving the other variables), which is denoted by $S = E(S)$, where S is the state of the program. One has the following rule,

$$\frac{p \Rightarrow q(E(S))}{\{p\}\, S = E(S)\, \{q\}} \tag{2}$$

Interpretation: If one wants q to hold after execution of the assignment statement, when S is replaced by $E(S)$, then $q(E(S))$ must hold before execution of the assignment; hence the precondition p must imply $q(E(S))$.

2. Sequence Rule: Let S be a sequence of two subprograms, say $S1$ and $S2$. One has the following rule,

$$\frac{\begin{matrix}\exists\, int: \\ \{p\}\, S1\, \{int\} \\ \{int\}\, S2\, \{q\}\end{matrix}}{\{p\}\, S1; S2\, \{q\}} \tag{3}$$

Interpretation: If there exits an intermediate predicate int that serves as a postcondition to $S1$ and a precondition to $S2$, then the conclusion is established.

3. Conditional Rule: Let S be a conditional statement, of the form: **if** (condition) **then** statement. One has the following rule,

$$\frac{\begin{matrix}\{p \land t\}\, S\, \{q\} \\ (p \land \neg t) \Rightarrow q\end{matrix}}{\{p\}\, if\ (t)\ then\, S\, \{q\}} \tag{4}$$

The interpretation of conditional rule is shown in Figure 4.1.

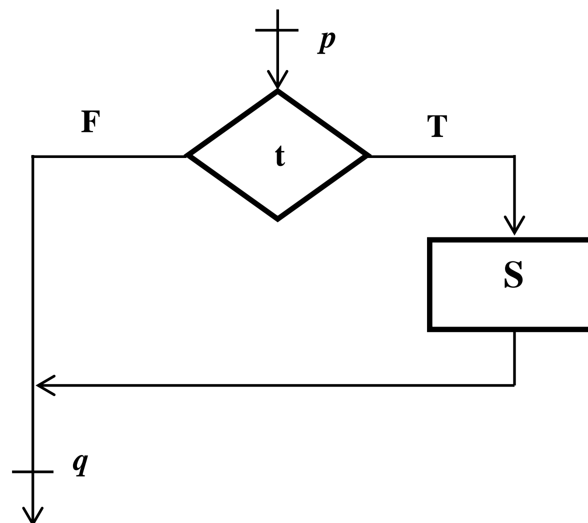


Figure 4.1: The Interpretation of Conditional Rule

4. Alternation Rule: Let S be an alternation statement, of the form: **if** (condition) **then** statement **else** statement. One has the following rule,

$$\frac{\begin{array}{c}\{p \wedge t\}\ S1\ \{q\} \\ \{p \wedge \neg t\}\ S2\ \{q\}\end{array}}{\{p\}\ if\ (t)\ then\ S1\ else\ S2\ \{q\}} \quad (5)$$

The interpretation of alternation rule is shown in Figure 4.2.

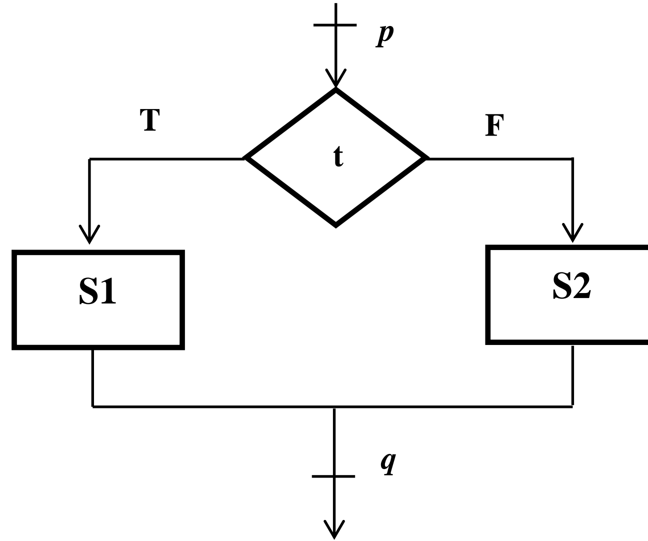


Figure 4.2: The Interpretation of Alternation Rule

5. Iteration Rule: Let S be an iterative statement, of the form: **while** (condition) statement. One has the following rule,

$$\frac{\begin{array}{c}\exists\ inv: \\ p \Rightarrow inv \\ \{inv \wedge t\}\ S\ \{inv\} \\ inv \wedge \neg t\ \Rightarrow q\end{array}}{\{p\}\ while\ (t)\ S\ \{q\}} \quad (6)$$

Interpretation: The first and second premises establish an inductive proof to the effect that predicate inv holds after any number of iterations. The third premise provides that upon termination of the loop, the combination of predicate inv and the negation of the loop condition must logically imply the postcondition. Predicate inv is called an invariant assertion. It must be chosen in order to be sufficiently weak to satisfy the first premise, yet sufficient strong to satisfy the third premise (and the second). See Figure 4.3 below, which highlight the points at which each of the relevant assertions is supposed to hold. Note that inv is placed upstream of the loop condition; hence the loop

condition is never part of inv (since upstream of the loop condition one does not know whether t is true or not).

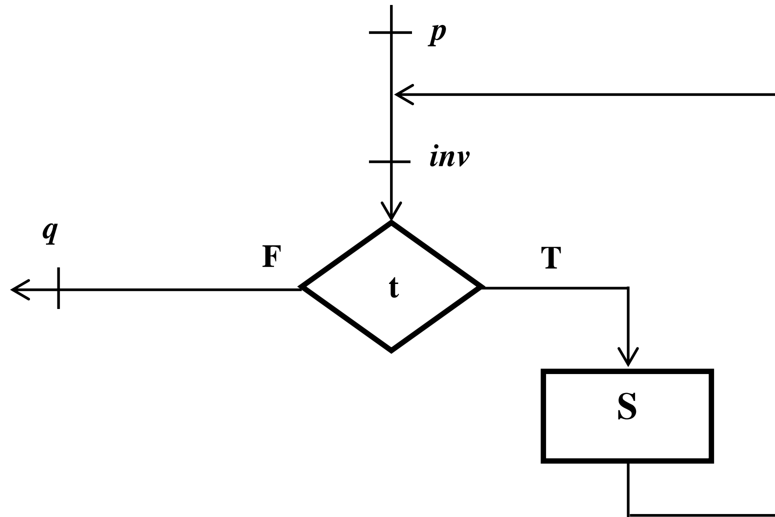


Figure 4.3: The Interpretation of Iteration Rule

6. Consequence Rule: Given a Hoare formula, one can always strengthen the precondition and/or weaken the postcondition. One has the following rule:

$$\frac{\begin{array}{c} p \Rightarrow p' \\ q' \Rightarrow q \\ \{p'\}\, S\, \{q'\} \end{array}}{\{p\}\, S\, \{q\}} \quad (7)$$

Interpretation: this rule stems readily from the definition of these formulas.

Using the proposed axioms and rules, one can generate theorems of the form $\{\boldsymbol{p}\}\ \boldsymbol{S}\ \{\boldsymbol{q}\}$. The following section presents illustrative examples of the inference system.

4.3.1.3 ILLUSTRATIVE EXAMPLES

We consider the following program on space S defined by variables x, y and z of type real, and we form a tripled by embedding it between a precondition and a postcondition:

- Program: $z = 0;\ while\ (y \neq 0)\ \{\ y = y - 1; z = z + x; \}$.
- Precondition: $x = x0\ \wedge y = y0$, for some constants $x0$ and $y0$.

- Postcondition: $z = x0 \times y0$.

The following formula is formed, and we attempt to prove that this formula is a theorem of the proposed inference system:

v: $\{x = x0 \wedge y = y0\}\ z = 0;\ while\ (y \neq 0)\{\ y = y - 1; z = z + x;\}\{\ z = x0 \times y0\}$

The sequence rule is applied to v , using the intermediate assertion $int \equiv (x = x0 \wedge y = y0 \wedge z = 0)$. This yields the following two formulas:

$v0$: $\{x = x0 \wedge y = y0\}\ z = 0\ \{(x = x0 \wedge y = y0 \wedge z = 0)\}$

$v1$: $\{(x = x0 \wedge y = y0 \wedge z = 0)\}\ while\ (y \neq 0)\ \{y = y - 1; z = z + x;\}\ \{z = x0 \times y0\}$

The assignment rule is applied to $v0$, which yields:

$v00$: $x = x0 \wedge y = y0 \Rightarrow (x = x0 \wedge y = y0 \wedge 0 = 0)$

It is found that $v00$ is a tautology, hence, it is an axiom of the inference system. The focus will be on $v1$, to which the iteration rule is applied, with the invariant assertion $inv \equiv (z + x \times y = x0 \times y0)$. This yields three formulas:

$v10$: $(x = x0 \wedge y = y0 \wedge z = 0) \Rightarrow (z + x \times y = x0 \times y0)$

$v11$: $\{y \neq 0 \wedge (z + x \times y = x0 \times y0)\}\ y = y - 1; z = z + x\ \{(z + x \times y = x0 \times y0)\}$

$v12$: $y = 0 \wedge (z + x \times y = x0 \times y0) \Rightarrow (z = x0 \times y0)$

It is found that $v10$ and $v12$ are both tautologies; hence, they are axioms of the inference system. Now consider $v11$, to which the sequence rule is applied, with the intermediate assertion $int \equiv (z + x \times (y + 1) = x0 \times y0))$. This yields two formulas:

$v110$: $\{y \neq 0 \wedge (z + x \times y = x0 \times y0)\}\ y = y - 1\ \{(z + x \times (y + 1) = x0 \times y0)\}$
$v111$: $\{(z + x \times (y + 1) = x0 \times y0)\}\ z = z + x\ \{(z + x \times y = x0 \times y0)\}$

The assignment rule is applied to $v110$ and $v111$, this produces:

$v1100$: $y \neq 0 \wedge (z + x \times y = x0 \times y0) \Rightarrow (\ z + x \times ((y - 1) + 1) = x0 \times y0)$

$v1110$: $(z + x \times (y + 1) = x0 \times y0) \Rightarrow ((z + x) + x \times y = x0 \times y0$

It is found that $v1100$ and $v1110$ are both tautologies; hence, they are axioms of the inference system. This concludes the proof to the effect that v is a theorem, since the sequence

$$v1110, v1100, v111, v110, v12, v11, v10, v00, v1, v0, v$$

is an inference, as the reader may check: each formula in this sequence is either an axiom, or the conclusion of a rule whose premises are to the left of the formula. It is concluded that the program

$$z = 0; while\ (y \neq 0)\ \{\ y = y - 1; z = z + x; \}$$

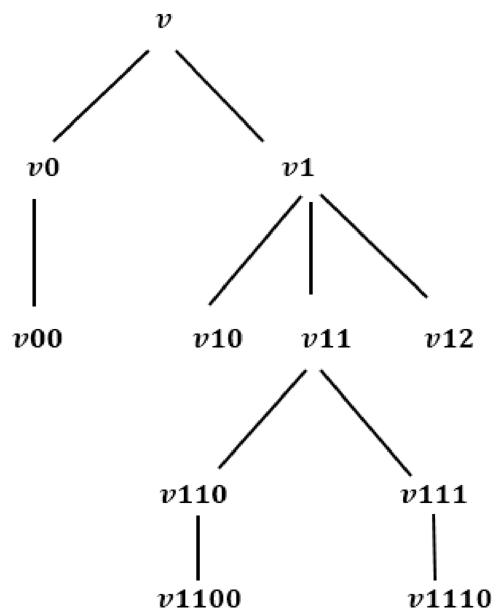
is partially correct with respect to the following specification:

$$R = \{(s, s') |\ \exists\ s0: x(s) = x(s0) \wedge y(s) = y(s0) \wedge z(s') = x(s0) \times y(s0)\}.$$

This formula can be simplified to be:

$$R = \{(s, s') |\ z(s') = x(s) \times y(s)\}.$$

It may be more illustrative to view this inference as a tree structure, where leaves are the axioms and internal nodes represent the rules that were invoked in the inference; the root of the tree represents the theorem that is established in the inference.



As a second example, consider the following Greatest Common Divisor (GCD) program on positive integer variables x and y:

$$while\ (x \neq y)\ \{\ if (x > y)\ \{x = x - y;\}\ else\ \{y = y - x;\}\}$$

and consider the following precondition/ postcondition pair:

- $p(S) \equiv (x = x0 \wedge y = y0)$,
- $q(S) \equiv (x = \gcd(x0, y0))$.

The following formula is formed:

v: $\{x = x0 \wedge y = y0\}$

$$while\ (x \neq y)\ \{\ if (x > y)\ \{x = x - y;\}\ else\ \{y = y - x;\}\}$$

$$\{x = \gcd(x0, y0)\}$$

The iteration rule is applied to v with the following invariant assertion: $inv \equiv (\gcd(x, y) = \gcd(x0, y0))$. This produces:

$v0$: $(x = x0 \wedge y = y0) \Rightarrow (\gcd(x, y) = \gcd(x0, y0))$

$v1$: $\{(\gcd(x, y) = \gcd(x0, y0)) \wedge (x \neq y)\}$

$$if\ (x > y)\{x = x - y;\}\ esle\ \{y = y - x;\}$$

$$\{(\gcd(x, y) = \gcd(x0, y0))\}$$

$v2$: $(\gcd(x, y) = \gcd(x0, y0)) \wedge (x = y) \Rightarrow (x = \gcd(x0, y0))$

It is found that $v0$ and $v2$ are tautologies, hence axioms of the inference system. The focus will be on $v1$, to which the alternation rule is applied, which yields:

$v10$: $\{(\gcd(x, y) = \gcd(x0, y0)) \wedge (x \neq y) \wedge (x > y)\}$

$$x = x - y$$

$$\{(\gcd(x, y) = \gcd(x0, y0))\}$$

$v11$: $\{(\gcd(x, y) = \gcd(x0, y0)) \wedge (x \neq y) \wedge (x \leq y)\}$

$$y = y - x$$

$$\{(\gcd(x, y) = \gcd(x0, y0))\}$$

Then the assignment statement rule is applied to $v10$ and $v11$, which yields:

$v100$: $(\gcd(x, y) = \gcd(x0, y0)) \wedge (x \neq y) \wedge (x > y) \Rightarrow (\gcd(x - y, y) = \gcd(x0, y0))$

$v110$: $(\gcd(x, y) = \gcd(x0, y0)) \wedge (x \neq y) \wedge (x \leq y) \Rightarrow (\gcd(x, y - x) = \gcd(x0, y0))$

It is found that both of these formulas are tautologies, hence axioms of the inference system. This concludes the proof that v is a theorem, since the sequence

$$v110, v100, v11, v10, v2, v1, v0, v$$

is an inference, as the reader may check: each formula in this sequence is either an axiom, or the conclusion of a rule whose premises are to the left of the formula. It is concluded that the program

$$while\ (x \neq y)\ \{\ if (x > y)\ \{x = x - y;\}\ else\ \{y = y - x;\}\}$$

is partially correct with respect to the following specification:

$$R = \{(s, s') |\ \exists\ s0: x(s) = x(s0) \wedge y(s) = y(s0) \wedge x(s') = \mathrm{gcd}(x(s0), y(s0))\}$$

This formula can be simplified to be:

$$R = \ \{(s, s') |\ x(s') = \mathrm{gcd}(x(s), y(s))\}.$$

4.3.2 FORMAL SOFTWARE VERIFICATION TECHNIQUES

Formal verification can be defined as a set of techniques used by developers to design systems that have reliable operation [50]. It complements testing and should be used in conjunction with it to increase reliability when developing a system. On the other hand, formal verification is regarded to be a better tool than testing because it is exhaustive and improves the knowledge of the system. However, an unfavorable feature of formal verification is that it is difficult and time-consuming [50]. Formal verification enhances the reliability of the developed system by making sure that it satisfies the defined functional requirements, especially in the first stages of design.

Formal verification is based on formal methods, which are mathematically based languages, techniques and tools for specifying and verifying systems [51]. There are various tools that aid in the processes of specification and verification. They provide notations and algorithms for system developers in order to formally specify and/or verify a system.

Formal Verification relies on the construction of a mathematical model of the system and on formally specifying the requirements to be checked against the system. Verification tools, the tools that perform formal verification, take two inputs: model of

a system and its specification and check if the system satisfies the specification. Based on how the modeling and the checking are carried out, the Formal Verification techniques are sub-divided into Static Analysis and Theorem Proving [50].

4.3.2.1 STATIC ANALYSIS

Static analysis encompasses a set of related techniques to automatically compute the information about the program behavior without executing it [52]. Most questions about the behavior of a program cannot be decided or their answers cannot be computed feasibly. Thus, the essence of static analysis is to efficiently compute approximate but sound guarantees: guarantees that are not misleading [52]. There are three static analysis techniques:

1. Abstract Static Analysis

Is techniques used in software development tools, for example in a pointer analysis in modern compilers. The formal basis for such techniques is the Abstract Interpretation, introduced by Cousot and Cousot [53].

"*Abstract Interpretation* is a theory of approximation of mathematical structures, in particular those involved in the semantic models of computer systems" [53].
"*An abstract domain* is an approximate representation of sets of concrete values" [52]. The function that is used to map concrete values to abstract ones is called an abstraction function. An abstract interpretation interprets the meaning of a program on an abstract domain to find an approximate solution [52].
An abstract interpretation can be derived from a concrete interpretation by defining counterparts of concrete operations, such as addition or union, in the abstract domain [52]. If specified mathematical constraints between the abstract and concrete domains are met, fixed points computed in an abstract domain are guaranteed to be sound approximations of concrete fixed points [54]. Abstract interpretation can be used for the systematic construction of methods and effective algorithms to approximate non-decidable or very complex problems in computer science. In particular, static analysis by abstract interpretation, which automatically infers dynamic properties of computer systems, has been recently very successful to automatically verify complex properties of real-time and safety-critical embedded systems [53].

Over the years, many and different abstract domains have been designed, specifically for computing invariants about numeric variables. The class of computable invariants, and hence of the provable properties, differs with the expressive power of a domain [52]. A static analyzer is thus parameterized by a *Numerical Abstract Domain*, i.e. a set of computer-representable numerical properties together with algorithms to compute the semantics of program instructions [50]. There are quite a few numerical abstract domains. Popular examples include the interval domain [53] that discovers variable bounds, and the polyhedron domain [55] for affine inequalities. Each domain achieves some cost versus precision balance. In particular, non-relational domains—e.g., the interval domain—are much faster but also much less precise than relational domains—able to discover variable relationships [50].

Shape analysis is a static code analysis technique that discovers and verifies properties of linked, dynamically allocated data structures in computer programs [50]. It is typically used at compile time to detect software bugs or to verify high-level correctness properties of programs. In Java programs, for instance, it can be used to ensure that a sort method correctly sort a list. For C programs, it is used to look for places where a block of memory is not properly freed [50]. Although shape analyses are very powerful, they usually take a long time to run. This is why they did not gain a widespread acceptance in places other than universities and research labs (where they are used but only experimentally) [50].

A first well-known static analysis tool for detecting simple errors in C programs is LINT. It was released in 1979 by Bell Labs. Numerous recent tools emulate and extend LINT in features including the kind of errors detected, warnings provided and user experience [52]. A notable example of such modern tools is FINDBUGS for Java. Grammatech Inc. produces CODESONAR tool, which uses inter- procedural analyses for checking template errors in C/C++ code. Kloc Work has K7 tool, which has similar features and supports Java. Coverity produces two analyzers: PREVENT has capabilities similar to those of CODESONAR, but also supports Microsoft COM and Win32 APIs and concurrency implemented using PThreads, Win32 or WindRiver VxWorks [52]. EXTEND is a tool for enforcing coding standards. The Astr´ee static analyzer [56] is abstract domains implemented for finding buffer overflows and undefined results in numerical operations. Astr´ee is used to verify the Airbus flight control software. Polyspace Technologies markets C and Ada static analyzers.

The experience of utilizing static analyzers can be compared with that of utilizing a compiler. The majority of analyzers are used to analyze large software systems with minimal user interaction. These tools are extremely robust i.e. they are suitable for large and varied inputs, and are efficient. In contrast, the properties required to be proved are often simple and are usually hard coded into the tools [52]. Unlike in model checking, generating counter examples is difficult or even impossible, due to the precision loss in join and widening operations [57].

2. Model Checking

Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model [50]. Generally, the check is carried out as an exhaustive state space search that is guaranteed to terminate because the model is finite. In model checking it is technically difficult to devise algorithms and data structures that enable handling large search spaces. Model checkers, tools for performing model checking, have two types of input: a finite state model of the system and a property specified formally [50]. A model checker functions to check if the system meets the property and gives either "yes" or "no" answer. If the answer is "no" (the system does not satisfy the property), model checkers will output a counter example (a system run that violates the property). Then this counter example can be analyzed to discover bugs in the system design [50].
There are many model checker we describe here the most popular ones. Holzmann's were pioneers in developing an explicit-state software model checking [58] [59]. Their SPIN project was used at the beginning for verification of temporal logic properties of communication protocols specified in the PROMELA language. Some software model checkers, such as the first version of the Java Pathfinder (JPF), translate Java code to PROMELA and use the SPIN for model checking [60]. Besides SPIN and JPF, there are two prominent representatives of the class of explicit-state software model checkers: CMC [61] and Microsoft Research's ZING [62]. The VERISOFT software verification tool attempts to avoid the state explosion by discarding the states it visits [63]. Because states are not stored, they should be repeatedly visited and explored. This method is incomplete for transition systems that contain cycles. In addition, it is stateless and the depth of its search has to be limited to avoid non-termination [52].

Compared to theorem proving, model checking is completely automatic and fast and often produces an answer in a matter of minutes. Model checking can also be used to check partial specifications, and thereby provide useful information about a system's correctness even if that system has not been completely specified. In conclusion, model checking produces counter examples, which usually represent subtle errors in design, and thus can be used to aid in debugging. The main drawback of model checking is the state explosion problem [50].

3. Bounded Model Checking

One of the most commonly applied formal verification techniques in the semiconductor industry is the Bounded Model Checking (BMC). The technique succeeded because of the impressive capacity of propositional SAT solvers. BMC was devised in 1999 by Biere et al. [64] as a technique intended to complement BDD-based unbounded model checking. It is termed bounded because it explores only the reachable states within a bounded number of steps. In BMC, the design under verification is unwound k times and conjoined with a property to form a propositional formula, which is passed to a SAT solver [50]. The formula is satisfied if and only if there is a trace of length k that disproves the property. The technique doesn't give a conclusion if the formula is unsatisfied, as there may be counterexamples longer than k steps. Despite these drawbacks, the technique is successful, as it is able to identify unnoticeable bugs. A satisfying assignment to the propositional formula relates to a passage from the initial state to a state where the property is being violated. BMC is considered to be the best technique of finding shallow bugs, and it provides a full counterexample trace when finding a bug. It also supports the widest range of program constructions. This support includes dynamically allocated data structures. This is why BMC does not require built-in knowledge about the data structures that are maintained by the program. However, the completeness is only obtainable in cases of very shallow programs or programs that do not have deep loops [50].

There are several BMC implementations for software verification. The first implementation, in the field of a depth-bounded symbolic search in software, is carried out by Currie et al. [65]. One of the earliest implementations of BMC for C programs is CBMC [66], [67], made at CMU; it emulates a wide range of architectures as an environment for the design under test. The main application of

CBMC is checking the consistency of system-level circuit models given in C or SystemC with an implementation given in Verilog. There is another version of CBMC developed by IBM for concurrent programs [68]. NEC Research developed the F-SOFT as the only tool that implements an unwinding of the entire transition system [69]. Its features include a SAT solver customized for BMC decision problems. SATURN is a specialized implementation of BMC, customized to the properties it checks and designed to implement loop unwinding [70]. The EXE tool is also devised to bug-hunting [71] where it combines explicit execution and path-wise symbolic simulation to detect bugs in system-level software such as file system code.

4.3.2.2 THEOREM PROVING

Theorem proving is a technique by which both the system and its desired properties are expressed as formulas in some mathematical logic [51]. This logic is given by a formal system, which defines a set of axioms and a set of inference rules. Theorem proving is the process of reaching a proof of a property from the axioms of the system. Steps in the proof refer to the axioms and rules, and possibly derived definitions and intermediate lemmas. Although proofs can be constructed by hand, however the focus will only be on machine-assisted theorem proving.

Today, theorem provers are increasingly being used in the mechanical verification of safety critical properties of both hardware and software designs. Theorem provers can be roughly categorized into a range from highly automated, general-purpose programs to interactive systems with special-purpose capabilities.

Unlike model checking, theorem proving is able to deal directly with infinite state spaces. It depends on techniques such as structural induction to prove over infinite domains. By definition, interactive theorem provers require interaction with a human; therefore, the theorem proving process is slow and often error-prone. However, and as an advantage, in the process of finding the proof the human user often gains invaluable insight into the system or the property being proved [50].

SUMMARY

The most important lesson one learns from this chapter is that: "*the software testing can show the presence of errors, but not their absence*" (as E. W. Dijkstra said) and that proving proves the absence of faults, but not their presence (since when a proof

fails we cannot tell whether it is because the program is faulty or because the proof is poorly planned). In addition, in order to study program testing one should understand the meaning of program correctness. Program correctness means a program behaves according to the specification for all circumstances planned for in the specification. The chapter also describes the concept of proving correctness and differentiates between the total correctness and partial correctness. It goes on presenting one of the program verification methods used to proof correctness known as "Hoare Logic". Furthermore, it explains how to use the inference system of Hoare Logic to prove that a program is partially correct with respect to a specification that takes the form of a precondition/postcondition. It also gives some examples of proving programs correctness using Hoare Logic. Finally, the chapter introduces the concept of formal verification techniques and gives some examples of different verification tools.

CHAPTER FIVE
AUTOMATED VERIFICATION WITH HAHA

5.1 INTRODUCTION

Computer systems correctness is vital in the contemporary information society. Despite the fact that modern computer systems are composed of complex hardware and software components, verifying the correctness of the software part is often a greater problem than that of the underlying hardware [52]. It is known that manual inspection of complex software is error-prone and expensive, therefore tool support is required. There are many tools that attempt to discover design errors using test vectors by examining certain executions of a software system. Formal verification tools, on the other hand, are used to check the behavior of a software design for all input vectors. A large number of formal tools for checking functional design errors in hardware are available and widely used. In contrast, markets of tools that check software quality are still in a beginning stage. Software quality is currently being subjected to broad research [52] .

According to practical experience, using static analysis methods for proving the correctness of computer programs is tedious, unreliable, obscure and entirely impractical process. This is because Hoare logic principles are generally not well-understood. In addition the whole concept is believed by a number of scholars as only a strange part of computer science history despite its fundamental role in the constantly developing field of program verification [52].

No doubt that this critical state of affairs has numerous reasons. However, there is one reason or issue that has greatly led to this problem: verification tools used in applying Hoare logic. These tools, in many cases, consist of a pen and paper, making it a tedious task to verify a whole program using a pen and a sheet of paper. It is more difficult than checking the correctness of the very same program in a less formal way. Even worse, both approaches are considered to have similar chances of making a mistake. Performing the whole logical inference on paper, without using the computer (save, perhaps, in the matter of typesetting) supports the view that static analysis is an impractical verification tool. A direct way to solve this problem is to use automated formal verification software to facilitate checking the correctness of Hoare programs.

This chapter surveys tools that perform automatic software verification to detect programming errors or prove their absence [72]. The two tools considered are tools that based on Hoare logic namely, the KeY-Hoare and HAHA. A short tutorial on these tools is provided, highlighting their differences when applied to practical problems. In addition, we evaluate the two tools in order to select one of them for the verification purpose to reaches the objectives of this research.

5.2 TOOLS FOR FORMAL PROGRAM VERIFICATION

Formal program verification tools are generally classified into three categories: interactive or semi-automatic proof construction environments like Isabelle [73] or the Prototype Verification System [74], tools for model checking (using abstract interpretation and similar techniques) like the Symbolic Model Verifier [75], and systems based on the Hoare calculus or related approaches (weakest precondition, dynamic logic) like the Frege Program Prover [76], Perfect Developer [77], the KeY-Hoare [78] or HAHA [79].

Since we concentrate on structured program development using invariants, pre- and post-conditions, we selected the latter two systems for survey. Beneath, we give detailed descriptions of the selected two tools.

5.2.1 THE KEY-HOARE TOOL

KeY-Hoare [78] is built on top of the software verification tool KeY (which is one of the most powerful verification systems for Java developed jointly by groups at Universit¨ at Karlsruhe (Germany), Universit¨ at Koblenz-Landau (Germany), and Chalmers University of Technology (Sweden). It is distributed under the Gnu General Public License and features a Hoare calculus with state updates. The KeY-Hoare tool is available free of charge and can easily be installed. It is a verification system that utilizes a variant of Hoare logic with explicit state updates, which enable users to reason about a program correctness by means of symbolic forward execution. Differently, the assignment rule in more traditional Hoare logics requires less natural and harder to learn backwards reasoning. No explicit weakening rule is required and first-order reasoning is automated. The system is suitable for teaching program verification, because students can concentrate on reasoning about programs, after their

natural control flow and proofs are checked by the computer [78]. At the present time, the GUI of the KeY-Hoare tool contains several elements inherited from the full Java version. These elements are not useful in more specialized contexts, which should be firstly cleaned up and simplified. The current version of KeY-Hoare does not support arrays since Java arrays are too complicated [78]. It provides many features including partial correctness proofs, total and execution time correctness proofs and integer and Boolean typed arrays. It is worthy to note that the logic behind the tool is not pure Hoare logic, but Hoare logic with updates. Therefore, one difficulty, in the adoption of the tool, is that it is necessary to extend the Hoare logic formula to include the updates. Another difficulty with adoption of KeY-Hoare tool is that it exposes users to the KeY prove where numerous logic rules are applied at each step of verification. Users must be instructed to decide which of these logic rules should be applied and which of them is to be avoided in typical cases [79].

5.2.1.1 USING KEY-HOARE TOOL

KeY-Hoare is available from a website indicated in [80]. Besides compilation from the source code, providers offer a pre-compiled byte-code version and installation via Java Web start technology. Detailed installation instructions are also supplied by the website.

Input files for KeY-Hoare must have either **.key** or **.proof** as file extension. Conventionally, **.key** files contain only the problem specification, i.e., the program together with its specification. In contrast, **.proof** files include proofs (or proof attempts) and are created when saving a proof.

An example that illustrates the format is shown in Figure 5.1. An input file consists of three sections, these are:

1) A section starting with keyword \functions declares all required rigid function symbols used.
2) A section starting with keyword \programVariables declares all program variables used in the program. Local variables declarations within the program are not allowed. However, multiple declarations are permitted.

3) A section starting with keyword \hoare contains the Hoare triple with updates to be proven valid, i.e., it contains the program and its specification. The initial update usually contains an assignment of fixed but arbitrary logical values to the input variables of the program. As illustrated in Figure 5.1, we do not provide the update rule because we need the program to be similar to original Hoare logic.

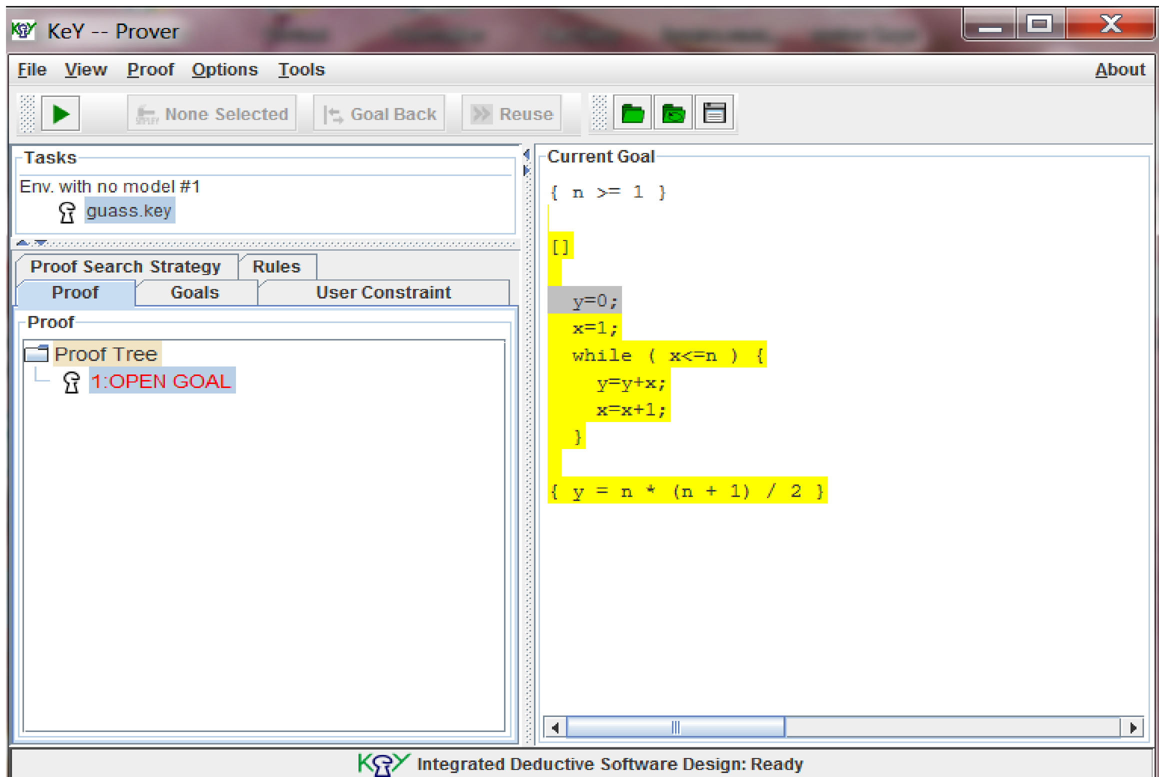


Figure 5.1: Screen Shot of KeY- Hoare System

After starting KeY-Hoare, the prove window becomes visible (the screenshot is displayed in Figure 5.1). The prove window consists of a menu- and toolbar, a status line and a central part split into a left and a right pane. The upper left pane displays a list of all loaded problems. The lower left pane offers different tabs for proof navigation or strategy settings. The right pane displays the currently selected sub-goal or an inner proof node. Before we explain the various sub-panes in more detail, our first task is to load a problem file. We can do this either by selecting Load in the File menu or by clicking on the icon that browse and load problem or proof files in the toolbar.

After loading the file, the right pane of the prove window displays the Hoare triple as specified in the input file. The proof tab in the left pane should display the proof tree

consisting of a single node. The first time during a KeY-Hoare session, when a problem file is loaded, the system loads a number of libraries. This loading takes a few seconds.

The upper part of the left pane displays all loaded problems while the lower part provides some useful tabs, these are:

- *The Proof tab* that shows the constructed proof tree. A left click on a node updates the right pane with the node's content (a Hoare triple with updates). Using a right click offers a number of actions like pruning, searching, etc.
- *The Goals tab* that lists all open goals, i.e., the leaves of the proof tree that remain to be built.
- *The Proof Search Strategy tab* that allows tuning automated proof search. The strategy for KeY-Hoare only allows adjusting the maximal number of rule applications before an interactive step is required, and (de-)activation of the auto-resume mode.

Excluding the above tabs, all other tabs are of no importance for KeY-Hoare. Therefore, they will be removed in future versions (as stated by the developer of KeY-Hoare).

The right pane displays the content of a proof node in two different modes. This depends on whether the node is (a) an inner node or a leave justified by an axiom or (b) it represents an open proof goal.

(a) The mode of "Inner Node View" is used for inner nodes of the proof tree. It highlights the formula which had been in focus at time of rule application as well as possible necessary side formulas. The applied rule is listed on the bottom of the view.

(b) The mode of "Goal View" is used when an open goal is selected. This view shows the Hoare triple to be proven and allows users to apply rules. Moving the mouse cursor over the expressions within the node highlights the smallest enclosing term or formula at the current position. A left click creates a popup window showing all applicable rules for the currently highlighted formula or term.

We illustrate how the system is used by proving correctness of a small program given in a number *n* returns the sum of all subsequent numbers from *1* to *n* inclusively. The code will look like the following as shown in Figure 5.2:

```
\functions {
}
\programVariables {
  int x;
  int y;
  int n;
}
\hoare{
{n >= 1}
\[{
        y = 0;
        x = 1;
        while (x <= n)
        {
                y = y + x;
                x = x + 1;
        }
}\]
{
  y= n * (n+1) / 2
}
}
```

Figure 5.2: KeY-Hoare Input File for the Gauss Example

All variables are integers. Provided that the starting value of *n* is nonnegative, the program always terminates with the value of *y* consistent with gauss formula (y= n * (n+1) / 2). A suitable precondition is *n* >= 1. The post-condition can be stated simply as y= n * (n+1)/2. The initial Hoare triple with updates reads as follows:

{n >= 1} [ ] y = 0; x = 1; while (x <= n) {y = y + x; x = x + 1;} { y= n * (n+1) / 2}

A file with an initial Hoare triple as proof obligation should be first loaded to the KeY-Hoare system. Then users can select a rule offered in a popup-menu after moving the mouse pointer over a Hoare triple and clicking. There is exactly one applicable rule for each program construct and the system offers typically this rule: users experience statement-wise symbolic execution of the target program. The only non-trivial interaction is to supply an invariant in a dialogue box that opens when the loop rule is applied. The invariant y = x* (x-1) / 2 & x<= n+1 & n >= 1 is sufficient.
Whenever reaching first-order verification conditions, the system offers a rule Update Simplification that applies the update rules automatically. At this point, users can opt to push the green Go button. Then the built-in first-order theorem prover tries to

establish validity automatically. If no proof is found, typically, the invariant or the specification (or the code!) is too weak or simply wrong. Inspecting the open goals usually gives users a good hint. The system allows users to follow symbolic execution of the program and to concentrate on getting invariants and specification right. First-order reasoning is thus left to the system. It is possible for users to inspect and undo previous proof steps as well as to save and load proofs.

Regarding the automation, up to this point the required interactive steps consisted of manual application of program rules and invocations of the strategies to simplify/ prove pure first-order problems. In order to avoid starting the strategies manually users can activate the auto-resume mode. This invokes the strategies on all open goals after each manual rule application and simplifies them as far as possible. In standard mode, they will not apply program rules. While performing a proof it is possible to save the current state at any time and to load it afterwards. For this users have to select File -> Save in the file menu and enter a file name ending with **.proof**.

5.2.2 HOARE ADVANCED HOMEWORK ASSISTANT (HAHA) TOOL

HAHA is a programming language firmly fixed in the new program development environment based on Eclipse [79]. Its purpose is to enable students learn Hoare logic. A programmer can write simple programs and annotate them with Hoare logic assertions. Then the environment verifies the assertions against the code and discharges them with help of external theorem provers, both automated and interactive. Users can write programs that manipulate integers and arrays. HAHA supports proofs of both partial and total correctness. HAHA relies on a SMT solver to prove the validity of generated formulae. Currently the only supported solver is Microsoft Z3 [81].

In the user interface of HAHA, shown in Figure 5.3, the main pane of the window is filled with the source code of the program that users work with. The interface shows an editor for simple while programs. It has all features expected from a modern IDE including syntax highlighting, automated completion proposals and error markers. Once a user enters a program, it is processed by a verification condition generator that implements the rules of Hoare logic. The resulting formulae are then passed to an automated prover. If the solver is unable to specify the correctness of the program, error markers are generated to direct users to the assertions, which could not be

proven. A very useful feature of HAHA is its ability to find counterexamples for incorrect assertions. These are included in error descriptions displayed by the editor. The input language of HAHA is that of while programs over integers and arrays. It's designed so that its mechanisms and data types match those supported by state of the art satisfiability solvers, e.g. Z3 [81] or CVC4 [82].

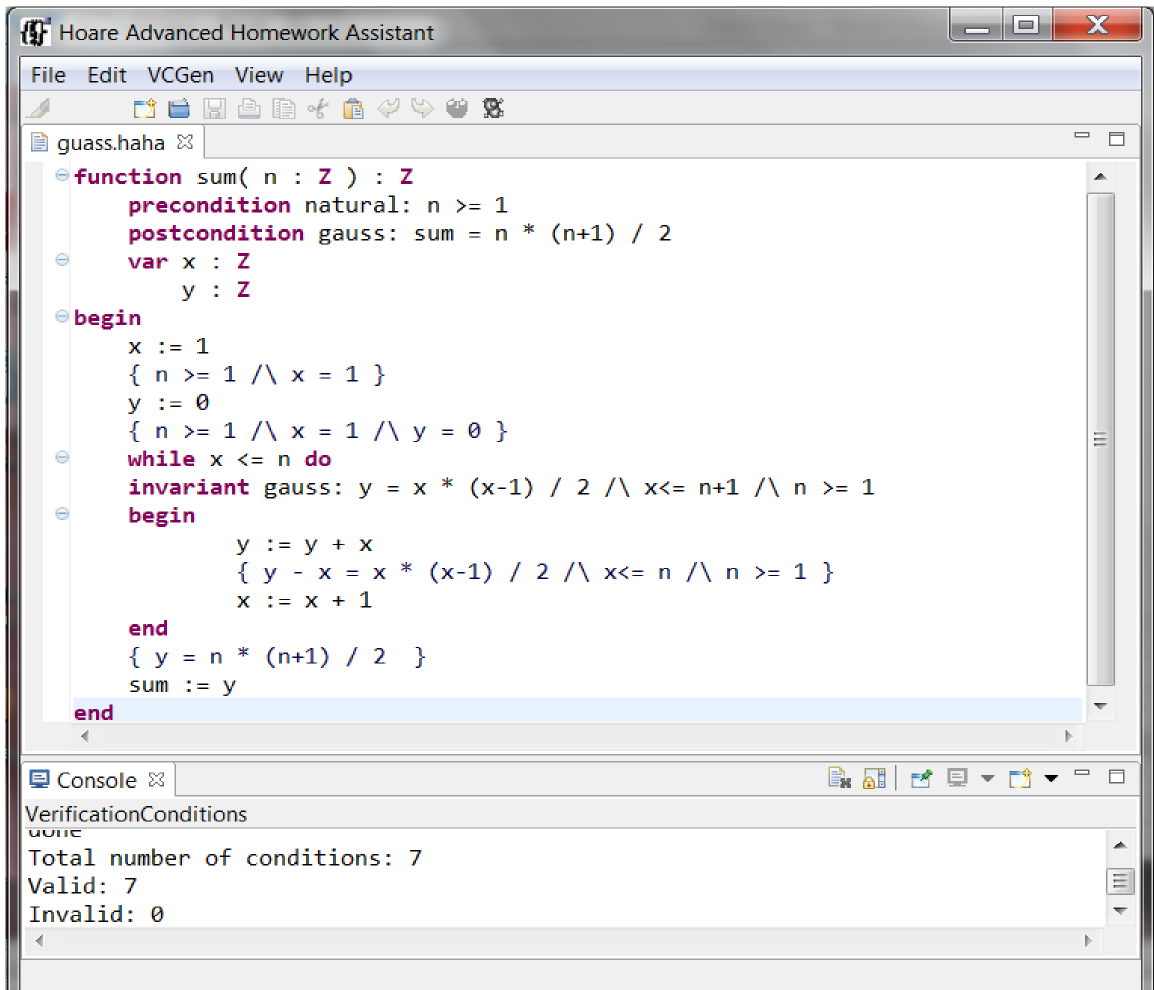


Figure 5.3: Screen Shot of HAHA System

A notable strength of HAHA is that the syntax of the programs is close to original Hoare logic with possible extensions, but in a manner that is easily digestible by users who are familiar with programming languages such as Pascal or Java. Added to that, the process of verification gives the impression that it is carried out as part of program development, in particular, the environment does not change to the one of interactive prover in order to assist in discharging verification conditions. Instead, users give assertions between instructions that are subsequently verified by an automatic theorem

prover [79]. Another strength of HAHA is that it is easily adaptable to any teaching environments.

In contrast to the common trend to make only loop invariants obligatory, HAHA compels students to fill in all intermediate assertions. This might seem surprising, because the requirement to write many formulae increases the amount of work necessary to create a verified program. However, in the case of a teaching aid, this approach is more beneficial. First, this suggests students to match the assertions with relevant Hoare logic rules. In this way, it reinforces the process of teaching the logic. Second, it gives the students a tangible experience of how much information must be maintained at each step of the program in order to execute it correctly— the process of making the verification work also gives students a tangible experience, making them see how it is easy to overlook transformation of some tiny detail in this information packet [79].

5.2.2.1 USING HAHA TOOL

To run HAHA, launch the haha executable from the installation directory. HAHA is typical file editor with standard and intuitive commands. It is important to note that source files should have the extension **.haha** in order for the editor to function properly. When the correct file extension was used, the code should be highlighted. To start the verification process, a user right-clicks anywhere in the editor and chooses "**Generate VCs"** from the displayed menu. This command can be also accessed from the main menu and the toolbar. HAHA will then present a console with computed verification conditions and messages logged during verification. If there were any problems, error markers will be added to the editor.

Suppose that a user wants to write a function that given a number of *n* returns the sum of all subsequent numbers from *1* to *n* inclusively. The code will look like the following as shown in Figure 5.4 [79]:

```
function sum( n : Z ) : Z
precondition natural: n >= 1
postcondition gauss: sum = n * (n+1) / 2
var x : Z
```

```
y : Z
begin
x := 1
{ n >= 1 /\ x = 1 }
y := 0
{ n >= 1 /\ x = 1 /\ y = 0 }
while x <= n do
invariant gauss: y = x * (x-1) / 2 /\ x<= n+1 /\ n >= 1
begin
y := y + x
{ y - x = x * (x-1) / 2 /\ x<= n /\ n >= 1 }
x := x + 1
end
{ y = n * (n+1) / 2 }
sum := y
end
```

Figure 5.4: HAHA Input File for the Gauss Example

The code should not be surprising, as users have all seen at least one implementation of the sum in their lives, especially for those who are familiar with Pascal as much of the syntax is based upon the language.

We start with a header of the function: **function** sum (n: **Z**): **Z** contains the keyword *function* that tells the system to interpret the following expressions as a function. Then the name of the function is given, in this case it is sum. The information about its parameters is enclosed in the parentheses. In our case, users have one formal parameter that is called *n*. Users declare the type of the parameter after the colon. It is Z this time, i.e. the type of integer numbers, as users know them from mathematics (these are not 32-bit integer numbers frequently met in programming languages). At the end, the parenthesis with parameters is followed by the declaration of the result type. In our case again this is Z. users can follow this function header by the definition of the body.

All variables that are used in code must be declared beforehand. Therefore, the declarations of the variables *x* and *y* are added between the function header and body. The keyword *var*, as in Pascal, marks the beginning of variable declaration sequence.

Contrary to Pascal, the elements of the sequence are not separated with semicolons, but with newlines. This holds for both variable declarations and instructions. The variable assignment is made in Pascal style as in x: = 1. We also use the Pascal style to define the return result of the function, i.e.: sum: = y. The while loop is defined by a phrase of the form:

while x <= n **do** followed by an instruction the loop iterates over. In our case this is a block instruction enclosed between begin and end, i.e.:

begin

y := y + x

x := x + 1

end

To express our intent with regard of the function, we can add pre-condition and post-condition formulas. These are located between the function header and its body. In this case the pre-condition, introduced with *precondition* keyword, says that the function can be called when the parameter n is not less than 1. The post-condition, introduced with *postcondition* keyword, says that the result is equal to the commonly known Gauss formula i.e. closed formula for the sum of the integers from 1 to n. These conditions can be named to make future reference easier:

precondition natural: n >= 1

postcondition gauss: sum = n * (n+1) / 2

In Hoare logic each instruction must be surrounded by two assertions. The first one describes the condition of the program state expected before the instruction and the second one describes the state resulting from the execution of the instruction. The precondition-postcondition pair is the assertions for the whole function. The assertions for other instructions are written in curly brackets located between instructions. To save notational burden, we do not write the first and last assertions in function since these are expressed with precondition and postcondition respectively. Therefore, the initial assignments decorated with the assertions look as follows:

begin

x := 1

{ n >= 1 /\ x = 1 **}**

y := 0

{ n >= 1 /\ x = 1 /\ y = 0 **}**

while x <= n **do**

The loop invariant condition, i.e. the formula that at the entry point to the loop at each its iteration, is marked with a special keyword *invariant*. So the while loop header augmented with the invariant is as follows:

while x <= n **do**

invariant gauss: y = x * (x-1) / 2 /\ x<= n+1 /\ n >= 1

We can name this invariant formula to make future reference more accurate. Again, the presence of the invariant formula gives us excuse not to mention assertions at the beginning and at the end of loop body as they equal to the invariant.

5.3 EVALUATION

We evaluated the tools with respect to the following criteria [72]:

1. ***Ease of use:*** Users should not be forced to spend too much time on learning a new language and on understanding its characteristics. Moreover, the system should be able to discharge the proof obligations as automatically as possible.
2. ***Feedback on errors:*** Error messages and reports on proof failures should provide readily understood feedback without deep knowledge of the tool specific underpinning formalism. In this way, users keep the focus on the verification problem itself.
3. ***Adequate documentation:*** There should be sufficient documentation that is easily understood by users. The better the documentation the less help is needed on basic language issues.
4. ***Ease of installation:*** The system should be easy to install on any of the platforms users typically use, i.e., on Windows, Linux, and Mac OS. In addition to easy installation, further software components from other sources should not be needed in installation.

In the remaining section, we describe our experiences with each of the two tools [72]:

1. **The KeY-Hoare Tool:** The evaluation was performed with version 0.1.9 of KeY-Hoare. Installation under Windows, Linux and MacOS is straightforward. The first thing we note about KeY-Hoare is its simplicity: given a program, pre-condition and post-condition, the system checks whether the program is correct with respect to its specification. However, the proofs are not fully automatic, and human

intervention is needed, as users must select the appropriate rule at each statement. Another reason for simplicity is that users who are acquainted with C++ will learn KeY-Hoare in a short time; the web site of KeY-Hoare offers sufficient documentation.

The work with KeY-Hoare exhibited some practical problems. KeY-Hoare assumes that the program is syntactically correct and well typed or the program will not be loaded to the system. Another problem is that the logic behind the KeY-Hoare tool is not pure Hoare logic, but Hoare logic with updates. Therefore, the first obstacle in the adoption of the tool was the need to add the updates formula to the original Hoare triple.

2. **The HAHA Tool:** The evaluation was performed with version 0.5 of HAHA that is tested with Z3 version 4.3.0. HAHA relies on a SMT solver to prove the validity of generated formulae. Currently the only supported solver is Microsoft Z3. Installation is available under many platforms such as Windows, Linux and MacOS. Sufficient documentation is available in HAHA web site and a great support is offered by the developer of HAHA when one requests help.

   HAHA supports proofs of partial and total correctness both automated and interactive. Users can write programs that manipulate integers and arrays. In addition, HAHA allows users to define Boolean variables but the verification condition generation is not supported yet for Boolean type. HAHA has the ability to find counterexamples for incorrect assertions when the solver is unable to ascertain the correctness of the program.

   One of the strengths of HAHA is that the syntax of the programs is close to original Hoare logic. However, HAHA forces users to fill in all intermediate assertions between instructions. This step increases the amount of work necessary to create a verified program, since the users need to write many formulae. Another notable strength of HAHA is that the tool is easily digestible by users who are familiar with programming languages such as Pascal or Java.

For Hoare-based verification purpose, HAHA is a better choice than KeY-Hoare tool at present. It offers a high-level language that can be learned within a short time. In addition, the high degree of proof automation combined with the ability to provide feedback on failed proof attempts enable users to easily prove the correctness of

programs. Therefore, HAHA removes the difficulty associated with the process of applying Hoare logic manually.

SUMMARY

This chapter surveys two main tools for automatic formal verification of software based on Hoare logic. It's focused on tools that provide some form of formal guarantee, and thus, aid to improve software quality. A short tutorial on these tools is provided, highlighting their differences when applied to practical problems. The chapter also evaluate the tools, provides the main features of each tool, and shows that HAHA is a better choice than KeY-Hoare tool because it offers a high-level language, which can be learned within a short time. Also, the high degree of proof automation together with the ability to provide feedback on failed proof attempts make users comfortable when using it to prove the correctness of their programs. Therefore, HAHA removes the difficulty associated with the process of applying Hoare logic manually.

CHAPTER SIX

ALNEELAIN SPECIFICATION LANGUAGE AND COMPILER

6.1 INTRODUCTION

In computer science, specification language is defined as a formal language used during systems analysis, requirements analysis and system design to describe a system [83]. This description will be at much higher level than programming, which is used to produce the executable code for a system [83]. It is important to note that specification languages are not directly executable and that they are meant to describe the "what", not the "how". If requirements are cluttered with unnecessary implementation details, this will be considered as an error. Formal specification language provides mathematical representation of the system and expresses the specification in a language whose vocabulary syntax and semantic are properly defined.

A formal specification language is usually composed of three primary components [84]:

1. *a syntax* that defines the specific notation with which the specification is presented,
2. *a semantic domain* that helps define a "universe of objects" that will be used to describe the system, and
3. *a set of relations* that define the semantic rules that indicate how objects may be manipulated properly and satisfy the specification.

Errors coming from numerous sources will crop up in specifications, just as they do in programs. A great advantage of formal specification is that tools can be used to discover and separate a large number of errors. Today, a variety of formal specification languages is in use such as OCL [85], Z [86], LARCH [87], and VDM [88]. Some of them are based either on algebraic specification or on model-based specification.

In this chapter, the design of a new specification language will be discussed. It is based on axiomatic specification proposed by Mili and Tchier [7] (discussed in detail in **Chapter 3**). This specification language is called *Alneelain* specification language [89]. *Alneelain* is considered as a mere extension of the axiomatic specification. The

reason that we need *Alneelain* is that the axiomatic notation by itself is difficult to compile. So we add enough markers to make it easy to compile and easy to read. The primary concern in designing the compiler of *Alneelain* specification language was with syntax and semantics of the language (*the frontend* of the compiler). The paramount goal of this chapter is to explain methods for furnishing a precise definition of the syntax and semantics of a specification language.

6.2 ALNEELAIN SPECIFICATION LANGUAGE AND COMPILER

Writing a compiler is a non-trivial task. Therefore, it is a good idea to structure the work. This will be typically done by breaking down the process into several phases that have well-defined interfaces. Conceptually, these phases operate in sequence but in practice, they are often interleaved. Each phase (obviously excluding the first phase) takes the output of the previous phase as its input. Figure 6.1 shows the flowchart of the idea to design *Alneelain* specification language. The first three steps of the flowchart are discussed in details in **Chapter 3**; as a result, there is a list of axioms and rules for many ADT's. Here, the beginning will be by describing a meta-language for syntax specification called Backus-Naur Form (BNF) [90]. Then BNF will be used to define the syntax of *Alneelain* specification language. Finally, the lexical analyzer and syntax checker for the language will be written.

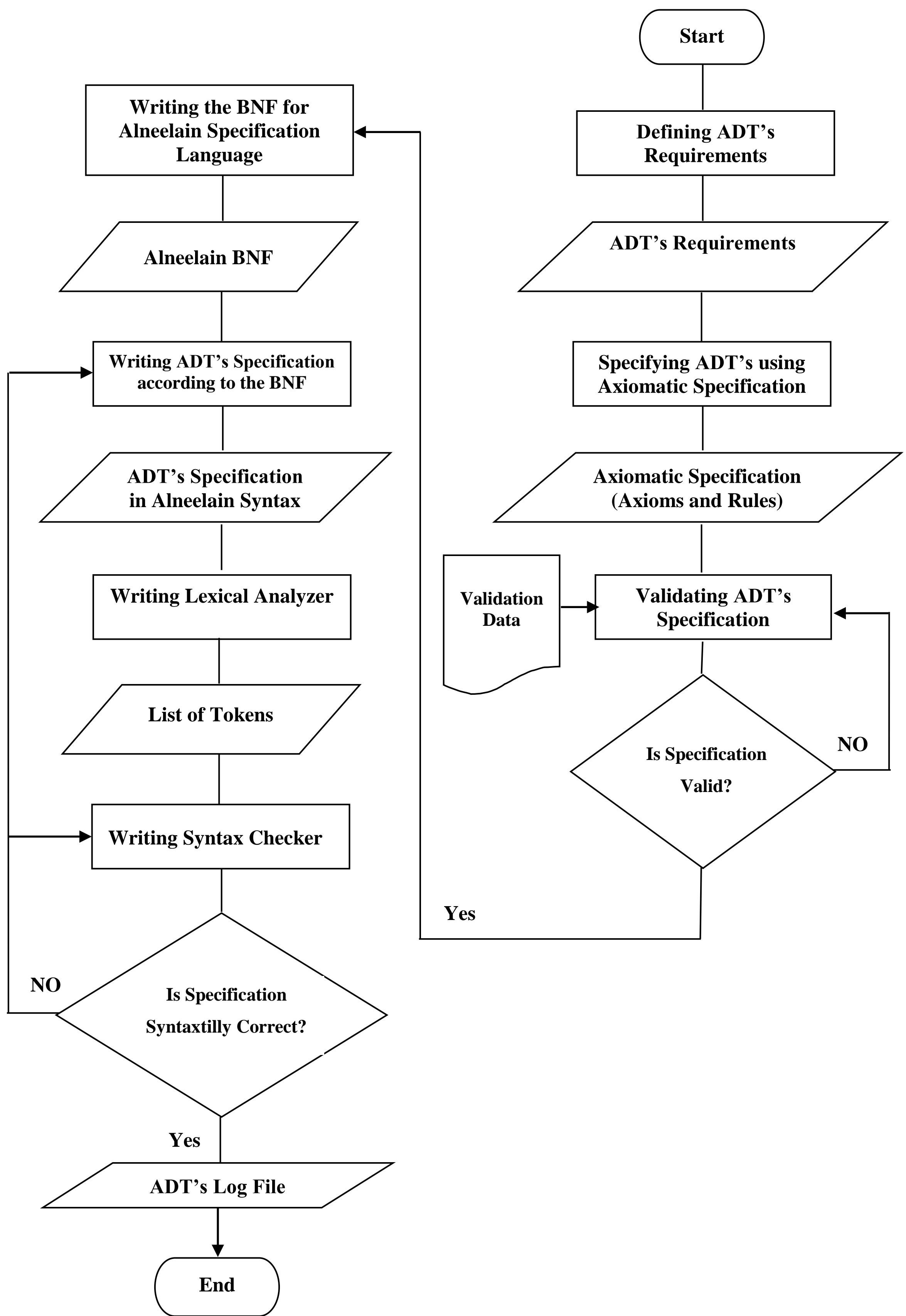


Figure 6.1: The Flowchart for Steps of Designing Alneelain Specification Language

6.2.1 GRAMMARS AND BNF FOR ALNEELAIN SPECIFICATION LANGUAGE

Formal methods were found to be more successful with describing the syntax of programming languages than with explaining their semantics [91]. Defining the syntax of programming languages bears a close resemblance to formulating the grammar of a natural language. Precisely, it is describing how symbols may be formed into the valid phrases of the language [91]. The formal grammars proposed by Noam Chomsky [92] for natural languages do apply to programming languages as well.

Definition: A grammar < **S, N, P, S** > consists of four parts:

1. A finite set **S** of terminal symbols, the alphabet of the language that are assembled to make up the sentences in the language.
2. A finite set **N** of nonterminal symbols or syntactic categories, each of which represents some collection of subphrases of the sentences.
3. A finite set **P** of productions or rules that describe how each non-terminal is defined in terms of terminal symbols and nonterminals. The choice of non-terminals determines the phrases of the language to which the meaning is ascribed.
4. A distinguished non-terminal **S**, the start symbol that specifies the principal category being defined for example, sentence or program.

According to the traditional notation for programming language grammars, nonterminals (that needs to be further expanded) are represented with the form "<category-name>" and productions as follows:

<Alneelain>::= <header>**;** <body> **endspecification**

where "header" and "body" are nonterminal symbols and "**endspecification**" and "**;**" are *terminal* (not enclosed in < > they represent themselves) symbols in the language. The symbol "::=" is part of the language for describing grammars and can be read "is defined to be" or "may be composed of ". There are many other symbols used in BNF to describe the language such as:

- The symbol '|' means *or;* it separates alternatives.
- Surround one or more symbols by […] to make them optional.
- Use {…} to surround a sequence of symbols that need to be treated as a unit (or group).

The grammar is described as “context-free” if only single non-terminals occur on the left sides of the rules. These grammars correspond to the Backus-Naur Form or BNF grammars. It is an important language for the researchers who used it first to describe Algol60 [90]. It plays a major role in defining programming languages. The BNF is defined as “a language for defining languages” i.e. it is a meta-language [91]. BNF greatly simplifies semantic specifications by formalizing syntactic definitions. Figure 6.2 shows the BNF for *Alneelain* specification language.

<Alneelain>::= <header>; <body> **endspecification**

<header>::= **specification** <specname>

<specname>::= < identifier>

<identifier>::= <letter> | <letter> < identifier>

<letter>::= **a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z**

|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<body>::= [<constsection>**;**]

[<typesection>**;**]

<inputsection>**;**

<outputsection>**;**

<varsection>**;**

<axiomsection>**;**

<rulesection>**;**

<constsection>::= **constant**<constbody>

<constbody>::= <constantdeclaration>

| <constantdeclaration> **,** < constbody>

<constantdeclaration>::= <constname> **=** <constvalue>

<constname>::= < identifier>

<constvalue>::= <digit> | <digit> <constvalue>

<digit>::= **0|1|2|3|4|5|6|7|8|9**

<typesection>::= **type** <typebody>

<typebody>::= <typedeclaration>

|<typedeclaration>**,** < typebody>

<typedeclaration>::= < typename> **:** <typedefinition>

< typename>::= < identifier>

<typedefinition>::= < identifier>

<inputsection>::= **input** <voppart> <ooppart> **endinput**

<voppart> ::= <vopdeclaration>

| <vopdeclaration> **,** <voppart>

<vopdeclaration>::=**vop**<methodname>[(<paramtype>[{,<paramtype>}])]**:**<returntype>

<ooppart> ::= **oop** <oopdeclarations>

<oopdeclarations> ::= <oopdeclaration>

| <oopdeclaration> **,** <oopdeclarations>

<oopdeclaration> ::= <methodname> [(<paramtype> [{,<paramtype>}])]

<methodname>::= <identifier>

<paramtype>::= <identifier>

<returntype>::= <identifier>

<outputsection> ::= **output** <outputlist> **endoutput**

<outputlist>::= <outputtype>

| <outputtype> **^** <outputlist>

<outputtype>::= < identifier>

<varsection> ::= **variable** <vardeclarations>

<vardeclarations> ::= <vardeclaration>

|<vardeclaration> **,** <vardeclarations>

<vardeclaration> ::= <variablename> **:** <variabletype>

<variablename> ::= <identifier>

<variabletype>::= < identifier>

<axiomsection> ::= **axioms** <axiomlist> **endaxioms**

<axiomlist> ::= <axiom>

| <axiom> **,** <axiomlist>

<axiom> ::= **axiom** <axiomname> **:** <axiombodies>

<axiombodies> ::= <axiombody>

|<axiombody> **&** <axiombodies>

<axiombody> ::= <specterm> **=** <literal>

<specterm> ::= <specname> **(**<history>**)**

<rulesection>::= **rules** <rulelist> **endrules**

<rulelist>::= <rule>

|<rule>**,** <rulelist>

<rule> ::= **rule** < rulename > **:** <rulebodies>

<rulebodies> ::= <rulebody>

| <rulebody> **&** <rulebodies>

<rulebody> ::= <spectrum> <operators> <spectrum>

<operators>::= **=>** | **=**

Figure 6.2: The BNF for Alneelain Specification Language

BNF should be precise and expressive for the language. For that, a specification for many ADT's according to the BNF of *Alneelain* is written to make sure that the designers' BNF is precise in describing their language and is able to go through other steps without errors. Figure 6.3 presents the stack ADT specification according to *Alneelain* BNF. Other ADT's specification is given in APPENDIX-D and https://sites.google.com/site/nahidahmedali/.

---

specification Stack**;**

constant

x = 5**;** *// it defined here just to show how we can define constants*

type

itemtype **:** char**;**

```
        input
                vop top: itemtype ,
                vop size: integer ,
                vop empty: Boolean
                oop init, pop, push(itemtype )
        endinput;
        output
                itemtype  ^ error  ^  Boolean  ^  integer
        endoutput;
        variable
                a: itemtype ,
                h: inputstar ,
                hprime: inputstar ,
                hplus: inputplus ;
        axioms
                axiom topAxioms:
                        Stack(init.top) = error &
                        Stack(init.h.push(a).top) = a ,
                axiom sizeAxiom:
                        Stack(init.size) = 0,
                axiom emptyAxioms:
                        Stack(init.empty)=true &
                        Stack(init.push(a).empty)=false
        endaxioms;
        rules
                rule initRule:
                        Stack(h.init.hprime)=Stack(init.hprime) ,
                rule initpopRule:
                        Stack(init.pop.h) = Stack(init.h) ,
                rule pushpopRule:
                        Stack(init.h.push(a).pop.hplus) = Stack(init.h.hplus) ,
                rule sizeRule:
                        Stack(init.h.push(a).size) =1+ Stack(init.h.size) ,
                rule emptyRules:
                    Stack(init.h.push(a).hprime.empty)=> Stack(init.h.hprime.empty)&
                     Stack(init.h.empty) => Stack(init.h.pop.empty) ,
                rule VopRules:
                        Stack(init.h.top.hplus)=Stack(init.h.hplus) &
                        Stack(init.h.size.hplus)=Stack(init.h.hplus) &
                        Stack(init.h.empty.hplus)=Stack(init.h.hplus)
        endrules;
endspecification
```

Figure 6.3: The Stack Specification in Alneelain Specification Language

6.2.2 LEXICAL ANALYSIS

Generally, the word "lexical" means, "pertaining to words". In programming languages, words are looked at as objects including names, numbers, keywords etc. Word-like entities like these are traditionally called *tokens* [93].

A *lexical analyzer*, also termed a *lexer* or *scanner*, will, as its input, take a string of individual characters and divide it into tokens. In addition, it will filter out whatever separates the tokens (the so-called *white-space*), i.e., lay-out characters (spaces, newlines etc.) and comments [93].

The primary objective of lexical analysis is to make the subsequent syntax analysis phase easier. Theoretically, the process carried out during lexical analysis can be made an integral part of syntax analysis, and in simple systems, this is indeed often practiced. However, there are reasons for keeping the two phases separate [93]:

- **Efficiency**: A lexer is able to do the simple parts of the work faster than the more general parser can. Moreover, the size of a system split in two phases may be smaller than a combined system. This seems paradoxical but, as it can be seen, there is a non-linear factor involved. This factor makes a separated system smaller than a combined system.
- **Modularity**: The syntactic description of the language need not be cluttered with small lexical details such as white-space and comments.
- **Tradition**: Languages are often designed with separate lexical and syntactical phases in mind. Therefore, the standard documents of such languages typically separate lexical and syntactical elements.

Specifications for lexical analysis are conventionally written using regular expressions i.e. an algebraic notation for describing sets of strings. The generated lexers are in a class of extremely simple programs called finite automata [93]. Some terminology pertaining to lexical analysis are these:

- ***A token***: a string of input characters. It is taken as a unit and passed on to the next phase of compilation.
- ***Lexemes***: the specific character strings that make up a token. For example: token: identifier, lexeme: pi, etc.

- ***Pattern*****:** a rule that describes a set of strings associated to a token. It is expressed as a regular expression and it describes how a particular token can be formed. For example: identifier: ([a-z]||[A-Z]) ([a-z]||[A-Z])*.

Table 6.1 presents the different tokens for *Alneelain* specification language and their lexemes and patterns.

Table 6.1: Tokens, Lexemes, and Patterns for Alneelain Specifcation Language

<table>
<tr><th>Token</th><th>Lexeme</th><th>Pattern</th></tr>
<tr><td>Keywords</td><td>specification, endspecification, type, constant, input, endinput, vop, oop, output, endoutput, variable, axioms, axiom, endaxioms, rules, endrules, rule</td><td>All reserved word or terminals of language.</td></tr>
<tr><td>Digits</td><td>0 – 9</td><td>Any numeric constants</td></tr>
<tr><td>Identifiers</td><td>A - Z , a - z</td><td>([a-z]||[A-Z]) ([a-z]||[A-Z])*. Any English latter (capital or small letters) repeated one or more times. These are words used to construct names like constant names , variables names, etc.</td></tr>
<tr><td>Operators</td><td>= , => , + , - , > , < , >= , <=</td><td>equalsign, impliessign, plussign, minussign, greatersign, lesssign, greaterequalsign, lessequalsign</td></tr>
<tr><td>Special characters</td><td>{ , } , (,) , ; , : , ^ , & , ,</td><td>lbraces, rbraces, lparen, rparen, semicolon, colon, unionsign, Ampersand, comma.</td></tr>
<tr><td>Whitespace</td><td>‘ ‘</td><td>Any white space</td></tr>
<tr><td>Newline</td><td>\n</td><td>Any new line</td></tr>
<tr><td>Tab</td><td>\t</td><td>Any tab</td></tr>
</table>

6.2.3 SYNTAX ANALYSIS

The purpose of lexical analysis is to split the input into tokens while the purpose of syntax analysis (or alternatively “parsing”) is to recombine these tokens. The combination of tokens is not back into a list of characters, but into something, that reflects the structure of the text [93]. This something is exactly the data structure and it is called the “syntax tree” of the text. As the name indicates, this is a tree-like structure. The leaves are the tokens found by the lexical analysis. If the leaves are read from left to right, the sequence is the same as in the input text. Hence, in the syntax

tree, it is important how these leaves are combined to form the structure of the tree. It is also important how the interior nodes of the tree are labelled. The syntax analysis finds the structure of the input text. In addition, it must reject invalid texts by reporting syntax errors [93].

Since syntax analysis is less local in nature than lexical analysis, more advanced methods are required. We, however, use the same basic strategy: A notation understandable by humans is transformed into a machine-like low-level notation suitable for efficient execution. This transformation process is called “parser generation” [93].

The notation used here for human manipulation is context-free grammars. It is a recursive notation for describing sets of strings and imposing a structure on each such string. This notation can - in some cases - be translated almost directly into recursive programs (recursive descent).

6.2.3.1 RECURSIVE DESCENT

A recursive descent parser is a top-down parser, so called because it builds a parse tree from the top (the start symbol) down, and from left to right, using an input sentence as a target as it is scanned from left to right [93]. The actual tree is not constructed but is implicit in a sequence of function calls. This parser uses a recursive function corresponding to each grammar rule (that is, corresponding to each non-terminal symbol in the language). For simplicity, one can just use the non-terminal as the name of the function. The body of each recursive function mirrors the right side of the corresponding rule. In order for this method to work, one must be able to decide which function to call based on the next input symbol. Perhaps the hardest part of a recursive descent parser is the scanning: repeatedly fetching the next token from the scanner. It is tricky to decide when to scan, and the parser doesn't work at all if there is an extra scan or a missing scan [93].

Recursive descent uses recursive functions to implement predictive parsing [93]. The crucial idea is that each non-terminal in the grammar is implemented by a function in the program. Each recursive function looks at the next input symbol to choose one of the productions for the non-terminal. The right-hand side of the chosen production is then used for parsing in the following way:

A terminal on the right-hand side is matched against the next input symbol. If they match, one moves on to the following input symbol and the next symbol on the right hand side, otherwise an error is reported.
A non-terminal on the right-hand side is handled by calling the corresponding function and, after this call returns, continuing with the next symbol on the right-hand side. When there are no more symbols on the right-hand side, the function returns. As an example, Figure 6.4 shows pseudo-code for a recursive descent parser for the grammar in Figure 6.2.

```
// The parse functions for handling <alneelain> ::=<header> ; <body> endspecification
FUNCTION alneelain( )
{
        SET diagnosis=true
        FUNCTION header( )
        FUNCTION checktoken(semicolon)
        FUNCTION body( )
        FUNCTION checktoken(endspecification);
        IF (diagnosis= true)
                PRINT “Syntactically Correct"
        ELSE
                PRINT “Syntactically Incorrect"
}
// The parse functions for handling <header>::= specification <specname>
FUNCTION header( )
{
        FUNCTION checktoken(specification);
        FUNCTION gettoken( );
}
// The parse functions for handling <body> ::=[< constsection>;] [<typesection>;]
// <inputsection>; <outputsection>; <varsection>; <axiomsection>; <rulesection>
FUNCTION body( )
{
        IF (token=constant)
        {
```

```
            FUNCTION constsection( );
            FUNCTION checktoken(semicolon);
        }
        IF (token=type)
        {
            FUNCTION typesection( );
            FUNCTION checktoken(semicolon);
        }
        FUNCTION inputsection( );
        FUNCTION checktoken(semicolon);
        FUNCTION outputsection( );
        FUNCTION checktoken(semicolon);
        FUNCTION varsection( );
        FUNCTION checktoken(semicolon);
        FUNCTION axiomsection( );
        FUNCTION checktoken(semicolon);
        FUNCTION rulesection( );
        FUNCTION checktoken(semicolon);
}
```

Figure 6.4: Recursive Descent Parser for Grammar in Figure 6.2.

The program in Figure 6.4 merely checks if the input is valid. It can easily be extended to construct a syntax tree by allowing the parse functions return the sub-trees for the parts of input that they parse. A complete code for syntax checker can be accessed at https://sites.google.com/site/nahidahmedali/.

There are two output from the syntax analysis step: (1) a message that tells whether the specification is correct or not and (2) a log file that contains all axioms and rules for a given specification, if the specification is correct.

6.3 THE USER INTERFACE OF ALNEELAIN SPECIFICATION LANGUAGE

To run *Alneelain* Specification Language, launch the executable file (.exe file) from the installation directory at https://sites.google.com/site/nahidahmedali/. When started, the application will appear as shown below in Figure 6.5:

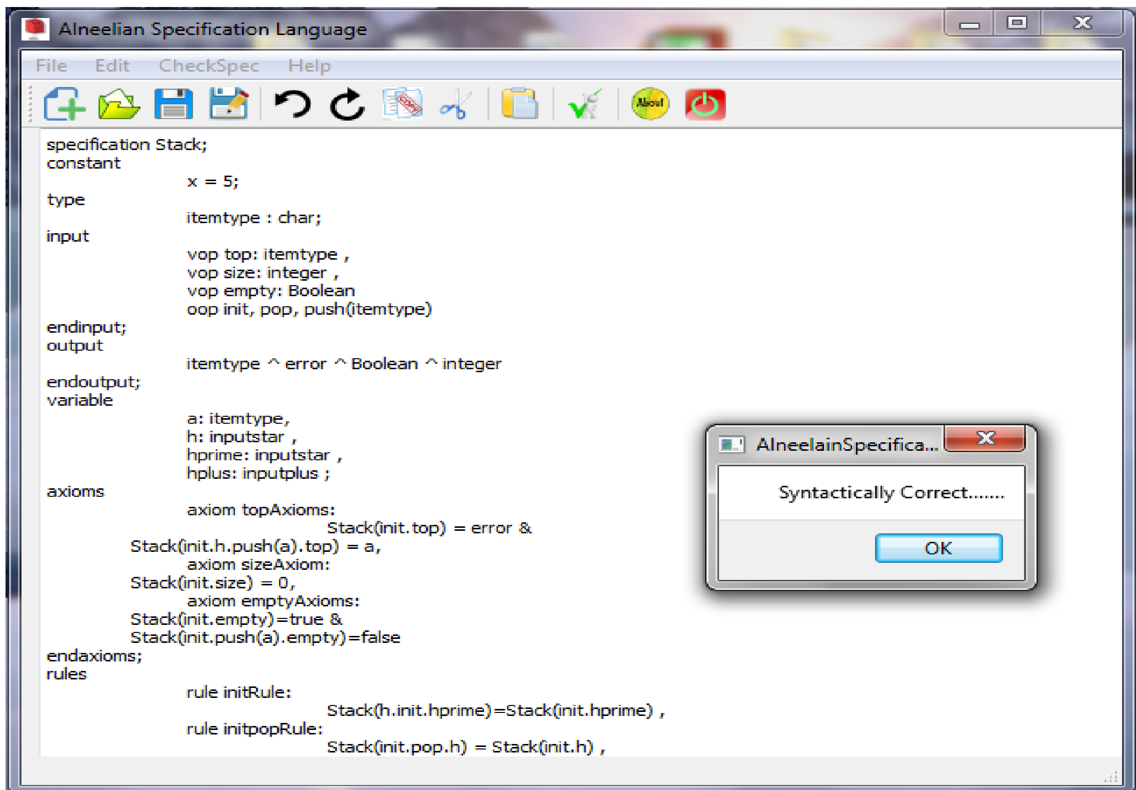


Figure 6.5: The User Interface of Alneelain Specification Language

The *Alneelain* Specification Language allows you to create a file that contains a specification of one abstract data type or any other specifications that can be written in the syntax of *Alneelain* language. In addition, it allows one to check if the specification is syntactically correct or not.

Alneelain Specification Language contains four main menus and one toolbar. The toolbar provides instant access to the most important commands from the menu. The four menus are:

1. File Menu:

Through the *File* menu, one can do the following:

New File: This item prompts you to create a file with an empty text editor. One can write the specification in the syntax of *Alneelain* Specification Language using the keyboard. The cursor will appear at current line in editor pane.

Open file: This item prompts you for the name and location of an existing *Alneelain* Specification Language file, and opens it in the application.

Save file: This item allows you to save the file that is open in the current tab of the editor window to its existing name. Note that the file should have the *.nln* extension.

Save file as: This item allows you to save the file that is open in the current tab of the editor window, but under a different name. Note that the file should have the *.nln* extension.

Exit: This item quits the application. It prompts you to save any unsaved files that are open in the editor.

2. Edit Menu:

This feature allows a selection of text to be copied or cut. Paste command places the text or object on the clipboard at the current cursor location in the currently active view or editor. Undo command reverses your most recent editing action and redo command re-applies the editing action that has most recently been reversed by the undo action.

3. CheckSpec Menu:

This feature causes the opened file to be processed. Checks for syntax are made; this can be used for quick check for simple errors. After performing a check, an alert box will appear that shows if the specification is syntactically correct or not. Suggestion and error messages may also appear.

4. Help Menu:

This menu provides help on using the *Alneelain* Specification Language application. Specifically, the help menu includes user manual and the about command which displays information about the application and installed features.

SUMMARY

In computer science, a formal specification language is defined as a specification language used during systems analysis, requirements analysis and system design to describe a system at much higher level than a programming. Formal languages are used because they allows users write specifications that are simultaneously precise, clear and concise. Numerous formal specification languages are in use at the present. This chapter discusses the design of a new specification language called *Alneelain*, which is based on axiomatic specification, and it goes

through the steps of designing *Alneelain* specification language. The design starts by a list of axioms and rules for many ADT's, then a meta-language for syntax specification called BNF is described, which plays a major role in defining syntax of programming languages. Then a specification for many ADT's according to the BNF of *Alneelain* is written to make confidence that the BNF is precise in describing the language. After that, the lexical analyzer is written which -as its input- will take a string of individual letters and divide this string into tokens. The chapter discusses the syntax checker for the language. The notation used for writing syntax checker is context-free grammar. It is a recursive notation that can be translated almost directly into recursive programs. Finally, the chapter displays the user interface for *Alneelain* specification language.

CHAPTER SEVEN

VERIFYING ADT IMPLEMENTATIONS AGAINST AXIOMATIC SPECIFICATIONS

7.1 INTRODUCTION

As seen from the previous chapters, this research focuses on the verification of ADT implementations against the axioms of the specification, using Hoare logic. Therefore, the advocated approach is based on three premises, namely:

- The specification of Abstract Data Types are written in a behavioral style, using a trace-like notation, which represents the behavior of an ADT by a relation from sequences of inputs to outputs.
- The specifications are defined inductively by means of axioms and rules, where the axioms represent the behavior of the ADT for elementary input sequences, and the rule equate the behavior of the ADT for complex input sequences to its behavior for simpler input sequences.
- Implementations are verified against axioms of the specifications using Hoare's logic; because axioms represent only part of the ADT behavior, verifying implementations against axioms is not sufficient. Therefore, as complement to this research, one can test implementations according to the cleanroom testing discipline [94], using the rules of the specification as oracles.

What makes Hoare logic difficult to apply on a large scale, and in particular, what makes it difficult to automate is the need to invent invariant assertions for iterative constructs in programs. However, our approach obviates this obstacle because axioms represent simple basic behavior of the ADT, they do not invoke complex calculations in the source code; at most, they may invoke some loops to initialize a data structure. Such code either uses loops for simple tasks, for which invariant assertions are simple to find, or does not invoke loops at all [95].

The present chapter discusses the approach that is taken to verify the correctness of ADT implementations against the axioms of specifications using Hoare logic and to give some examples of how to verify ADT implementations using this approach. The

chapter concludes with examples for verifying ADT implementations automatically using HAHA tool [79] .

7.2 DEFINING ADT'S SPECIFICATION

As discussed in **Chapter 3**, the specifications are defined inductively by means of axioms and rules, where the axioms represent the behavior of the ADT for elementary input sequences, and the rules equate the behavior of the ADT for complex input sequences to its behavior for simpler input sequences. An example of axioms for the stack data type include:

- **Size Axiom**

  stack(init.size) = 0.

- **Init Empty Axiom**

  stack(init.empty) = true.

- **Push Empty Axiom**

  stack(init.push(a).empty) = false.

- **Init Top Axiom**

  stack(init.top) = error.

- **Push Top Axiom**

  stack(init.h.push(a).top) = a.

7.3 IMPLEMENTATION OF ABSTRACT DATA TYPES

The implementation of ADT means the provision of one procedure or function for each abstract operation. According to the ADT's specifications, the ADT instances are represented by some concrete data structure that is manipulated by those procedures. Commonly there are numerous ways to implement the same ADT, using a number of different concrete data structures. Therefore, for instance, one can implement an abstract stack by a linked list or by an array. An ADT implementation is often packaged as one or more modules. The interface of such module contains only the signature (number and types of the parameters and results) of the operations. The implementation of the module — namely, the bodies of the procedures and the

concrete data structure used — can then be hidden from most clients in order to implement information hiding strategy. This information hiding strategy permits the implementation of the module to be altered without disturbing the client programs. For example, the assumption used is that the implementation of the stack ADT takes the form of a class in an object-oriented language. In this class each input symbol *(init, push, pop, top, size, empty*) represents a method of the class; whereas *init, push* and *pop* are void methods, *top* returns a value of the data type of the stack, *size* returns an integer and *empty* returns a Boolean. Figure 7.1 presents the user implementation of stack data type, it presents both *stack.h* and *stack.cpp* files. (This source code is taken from the Apache C++ Standard Library (STDCXX)).

```
//**********************************************************
//  Header file stack.h
//**********************************************************
const int SIZE = 100;
// an approximation of infinity: unbounded stack

typedef int sitemtype;
typedef int indextype;

class stack

   { public:

    stack();                // default constructor

    void init();       // initializes or re-initializes the stack

    bool empty () const;   // tells whether stack is empty

    void push (sitemtype sitem); // add sitem to the stack

    void pop ();                 // deletes top of the stack

    sitemtype top ();           // returns top of the stack

    int size ();                // returns size of the stack

    private:     //  array-based implementation.

    sitemtype sarray [SIZE];
    indextype sindex;
   };
// ************************************************************
// Array-based C++ implementation for the stack ADT.
// file stack.cpp, refers to header file stack.h.
// ************************************************************

#include "stack.h" // stack.h header file.

stack :: stack ()
```

```
{
    sindex=0;
}

bool stack :: empty ()  const
{
    return (sindex==0);
}

void stack :: init ()
{
     sindex   =  0;
}

void stack :: push (sitemtype sitem)
{
    sindex=sindex+1;
    sarray[sindex]=sitem;
}

void stack :: pop ()
{
    if (sindex >0) // stack is not empty
    {
           return sarray[ sindex-1];
    }
}

sitemtype stack :: top ()
{
    int error = -9999;
    if (sindex >0)
    {
       return sarray[sindex];
    }
    else
    {
        return error;
    }
}

int stack :: size ()
{
    return sindex ;
}
```

Figure 7.1: The Implementation of Stack Data Type

7.4 MAPPING AXIOMS TO HOARE FORMULAS

To each of the axioms presented in **Section 7.2**, a verification condition is associated in the form of a Hoare formula that must then be proved. The verification conditions generated for each of the axioms is shown below:

- **Size Axiom**

  *v: {**true**} init(); y=size() {y=0}*

  Where y is a variable of type integer.

- **Init Empty Axiom**

  *v: {**true**} init(); y=empty() {y=true}*

  Where y is a variable of type Boolean.

- **Push Empty Axiom**

  v*: {**true**} init(); push(a); y=empty() {y=false}*

  For an arbitrary item a, where y is a variable of type Boolean.

- **Init Top Axiom**

  *v: {**true**} init(); y=top() {y=error}*

  Where y is declared as a variable that can hold the data type of the items that we put on the stack (called itemtype) or the error message.

- **Push Top Axiom**

  *v: {**true**} init(); h; push(a); y=top() {y=a}*

  For an arbitrary item a, where y is a variable of type itemtype and *h* is an arbitrary sequence of operations.

7.5 VERIFYING ADT IMPLEMENTATIONS AGAINST AXIOMS

Abstract data type implementation is verified against axiomatic specification using Hoare Logic. In order to prove that each verification condition in the form of a Hoare formula associated to the axioms given above, each method call must be replaced by its body (source code). So that each *return* statement is replaced by *y* = the returned expression. Also, all the statements (in source code) are converted from C++ to Pascal-like notation in order to be able to summit to HAHA tool because it is based on Pascal. For example sindex= 0 is converted to sindex:=0 .This yield for example:

1. **Size Axiom:**

   v: {true}

   sindex :=0;

y:= sindex;

{y=0}

In order to prove formula *v*, the sequence rule is applied and the following formulas are generated using *{sindex=0}* as intermediate assertion:

v0: {true} sindex:=0 {sindex=0}

v1: {sindex=0} y:=sindex{y=0}

By applying the assignment statement rule to v0, v1, it is found that:

v00: true ⇒ 0=0,

v10: sindex=0 ⇒ sindex =0

v0, v1 are tautologies, hence axioms of Hoare logic. This concludes the proof, and establishes the validity of *v*.

2. Init Empty Axiom:

v: {true}

sindex :=0;

y:=(sindex=0)

{y=true}

In order to prove formula *v*, the sequence rule is applied and the following formulas are generated using *{sindex=0}* as intermediate assertion:

v0: {true} sindex: =0 {sindex=0}

v1: {sindex=0} y :=(sindex=0) {y=true}

By applying the assignment statement rule to v0, v, it is found that:

v00: true ⇒ 0=0,

v10: sindex=0 ⇒ (sindex=0) =true,

v0, v1 are tautologies, hence axioms of Hoare logic. This concludes the proof, and establishes the validity of *v*.

3. Push Empty Axiom:

v: {true}

sindex :=0;

sindex:=sindex+1; sarray[sindex]=:sitem;

y:=(sindex=0)

{y=false}

In order to prove formula *v*, the sequence rule is applied three times, and the following formulas are generated [96]:

v0: {true} sindex:=0 {sindex=0}

v1: {sindex=0} sindex:=sindex+1; {sindex=1}

v2: {sindex= 1} sarray[sindex]:=a; {sindex=1}

v3: {sindex=1} y:=(sindex=0) {y=false}

The assignment statement rule applied to v0, v1, v2, v3 yields, respectively:

v00: true ⇒ 0=0,

v10: sindex=0 ⇒ sindex+1=1,

v20: sindex=1 ⇒ sindex=1,

v30: sindex=1 ⇒ (sindex=0)=false,

All of which are tautologies, hence axioms of Hoare logic. This concludes the proof, and establishes the validity of *v*.

4. Init Top Axiom:

v: {true}

sindex:=0;

if (sindex>0) {y := sarray[sindex];} else {y:= error;}

{y=error}

In order to prove formula *v*, the sequence rule is applied and the following formulas are generated using *{sindex=0}* as intermediate assertion:

v0: {true} sindex:=0 {sindex=0}.

v1: {sindex=0} if (sindex>0) {y := sarray[sindex];} else {y:= error;}

{y=error }

By applying the assignment statement rule to v0, it is found that:

v00: true ⇒ 0=0

v0, is tautology, hence axioms of Hoare logic. v1 is considered, to which the alternation rule is applied. It is found that:

v10: {sindex=0 ∧ sindex>0} y := sarray[sindex]; {y=error}

v11: {sindex=0 ∧ sindex≤0} y := error; {y=error}

The formula v10 is vacuously valid since the precondition is false. The assignment statement rule is applied to v11:

v110: sindex=0 ⇒ error = error

This formula is a tautology, hence an axiom of Hoare's inference system; this concludes the proof.

5. **Push Top Axiom:**

v: {true}

sindex:=0;

h;

sindex:=sindex+1; sarray[sindex]:=a;

if (sindex>0) {y := sarray[sindex];} else {y:= error;}

{y=a}

Applying the sequence rule four times using intermediate assertion *{sindex≥0}* after sequence *h* [96] yields the following formulas:

v0: {true} sindex:=0 {sindex≥0}.

v1: {sindex≥0} h {sindex≥0}.

v2: {sindex≥0} sindex := sindex+1 {sindex>0}

v3: {sindex>0} sarray[sindex]:=a {sindex>0 ∧ sarray[sindex]=a}

v4: {sindex>0 ∧ sarray[sindex]=a}

if (sindex>0) {y := sarray[sindex];} else {y:= error;}

{y=a}

Application of the assignment rule to v0, v2, v3 yields, respectively:

v00: true ⇒ 0=0,

v20: sindex≥0 ⇒ sindex+1>0

v30: sindex>0 ⇒ sindex>0 ∧ a=a,

All of which are tautologies. v1, to which the consequence rule is applied, is considered:

v10: {sindex≥0} h {sindex≥0}.

In order to establish the validity of this formula, where *h* is a sequence of operations, it is sufficient (by virtue of induction) to prove it for each individual operation; it is certainly valid for V-operations, since by definition they do not modify state variables [96]. As for O-operations,

- Init: The formula *{sindex≥0} init() {sindex≥0}* is valid by virtue of the consequence rule, since *{true} init() {sindex=0}* is valid.
- Push: The formula *{sindex≥0} push(a) {sindex≥0}* is valid since the push operation increases the value of *sindex*.

- Pop: The formula *{sindex≥0} pop() {sindex≥0}* is valid since the pop operation does not decreases the value of *sindex* unless *sindex* is positive, and then it is only decreased by 1.

Now the focus is on formula v4, to which the alternation rule is applied. It is found that,

v40: {sindex>0 ∧ sarray[sindex]=a ∧ sindex>0} y := sarray[sindex]; {y=a}

v41: {sindex>0 ∧ sarray[sindex]=a ∧ sindex≤0} y := sarray[sindex]; {y=a}

The formula v41 is vacuously valid since the precondition is false. The assignment statement rule is applied to v40:

v400: sindex>0 ∧ sarray[sindex]=a ⇒ sarray[sindex]=a.

This formula is a tautology, hence an axiom of Hoare's inference system; this concludes the proof.

7.6 SUBMITTING TO HAHA THROUGH THE API

To verify the correctness of Hoare formulas automatically using HAHA tool, these formulas must be put in syntax of HAHA, which is discussed in detail at **Chapter 5**. Before start writing HAHA program, one thing to keep in mind is that the source files should have the extension **.haha** or the editor might not function properly. As examples:

1. **Size Axiom:** *v: {**true**} init(); y=size() {y=0}*
   v: {true}

sindex :=0;

y:= sindex;

{y=0}

<u>SizeAxiom.haha:</u>

```
function S() : Z
  var
    sindex : Z
    y : Z
begin
  skip
  {true}
  sindex:=0
  {sindex = 0}
  y:=sindex
```

```
  { y = 0 }
  skip
end
```

In HAHA, a function must be declared to start the proving using keyword *function* followed by the function name *S*( ). The result type is declared after the colon. It is Z this time, i.e. the type of integer numbers as known from mathematics. All variables should be declared beforehand using *var* keyword in this example variables are *sindex* and *y* and they are of type *Z*. *Skip* is necessary since HAHA does not allow assertions right after begin statement. {true} is the initial assertion (or precondition). Then comes the first statements (sindex: = 0). HAHA requires that all intermediate assertions are present in this example is {sindex=0}. After the last statement, there should be the final assertion (or postcondition) in this case is {y = 0}. *Skip* is necessary since HAHA does not allow assertions right before end. Finally the HAHA program is finished with keyword *end*. After this program is written one can check its correctness with Z3 checker. It is launched by choosing menu option VCGen → Generate Verification Conditions. HAHA will then present a console with computed verification conditions and messages logged during verification. Such as:

```
Total number of conditions: 3
Valid: 3
Invalid: 0
Unknown: 0
Errors: 0
Missing: 0
```

It is clear that the verification conditions are valid. If there were any problems, error markers will be added to the editor.

2. **Init Empty Axiom:** *v: {**true**} init(); y=empty() {y=true}*

v: {true}

sindex :=0;
y:=(sindex=0)

{y=true}

In principle HAHA allows one to define Boolean variables as follow:

var

boolvar : BOOLEAN

and then use constants true, false and logical expressions to set the value. However, the verification condition generation is not supported yet for this type due to the lack of resources on HAHA side as stated by it is developer. For that, integer variables are used instead and are checked for their particular values.

InitEmptyAxiom.haha:

```
function S() : Z
  var
    sindex : Z
    y : Z
begin
  skip
  {true}
  sindex:=0
  {sindex = 0}
  if (sindex=0) then
  y:=1
  else
  y:=0
  { y= 1 }
  skip
end
```

HAHA Result:

```
Total number of conditions: 4
Valid: 4
Invalid: 0
Unknown: 0
Errors: 0
Missing: 0
```

3. **Push Empty Axiom:** v*: {**true**} init(); push(a); y=empty() {y=false}*

   v: {true}

sindex :=0;

sindex:=sindex+1; sarray[sindex]=:sitem;

y:=(sindex=0)

{y=false}

PushEmptyAxiom.haha:

```
function S() : Z
  var
    sindex : Z
    sarray: ARRAY[Z]
    y : Z
    a :Z
```

```
begin
  skip
  {true}
  sindex:=0
  {sindex = 0}
  sindex:=sindex+1
  {sindex = 1}
  sarray[sindex]:=a
  {sindex = 1}
  if (sindex=0) then
    y:=1
 else
    y:=0
 { y = 0 }
  skip
end
```

HAHA Result:

```
Total number of conditions: 6
Valid: 6
Invalid: 0
Unknown: 0
Errors: 0
Missing: 0
```

4. Init Top Axiom: *v: {**true**} init(); y=top() {y=error}*

v: {true}

sindex:=0;

if (sindex>0) {y := sarray[sindex];} else {y:= error;}

{y=error}

InitTopAxiom.haha:

```
function S() : Z
  var
    sindex : Z
    sarray: ARRAY[Z]
    y : Z
    error: Z
begin
  skip
  {true}
  sindex:=0
  {sindex = 0}
  if (sindex>0) then
     y := sarray[sindex]
  else
     y:= error
```

```
  { y = error }
  skip
end
```

HAHA Result:

```
Total number of conditions: 4
Valid: 4
Invalid: 0
Unknown: 0
Errors: 0
Missing: 0
```

5. **Push Top Axiom:** *v: {**true**} init(); h; push(a); y=top() {y=a}*

v: {true}

sindex:=0;

h;

sindex:=sindex+1; sarray[sindex]:=a;

if (sindex>0) {y := sarray[sindex];} else {y:= error;}

{y=a}

In order to establish the validity of this formula, where *h* is a sequence of operations, it is sufficient (by virtue of induction) to prove it for each individual operation; it is certainly valid for V-operations, since by definition they do not modify state variables [96]. As for O-operations (init, push and pop) respectively:

1. **Push Top Axiom(init):** *v: {**true**} init(); init(); push(a); y=top() {y=a}*

v: {true}

sindex:=0;

sindex:=0;

sindex:=sindex+1; sarray[sindex]:=a;

if (sindex>0) {y := sarray[sindex];} else {y:= error;}

{y=a}

PushTopAxiom-init.haha

```
function S() : Z
  var
    sindex : Z
    sarray: ARRAY[Z]
    y : Z
    a : Z
    error: Z
  begin
  skip
```

```
  {true}
  sindex:=0
  {sindex = 0}
  sindex:=0
  {sindex = 0}
  sindex:=sindex+1
  {sindex = 1}
  sarray[sindex]:=a
  {sindex = 1 /\ sarray[sindex]=a }
  if (sindex>0) then
     y := sarray[sindex]
  else
     y:= error
  { y = a }
  skip
end
```

<u>HAHA Result:</u>

```
Total number of conditions: 7
Valid: 7
Invalid: 0
Unknown: 0
Errors: 0
Missing: 0
```

2. **Push Top Axiom(push):** *v: {**true**} init(); push(); push(a); y=top() {y=a}*

v: {true}

sindex:=0;

sindex:=sindex+1; sarray[sindex]:=a;

sindex:=sindex+1; sarray[sindex]:=a;

if (sindex>0) {y := sarray[sindex];} else {y:= error;}

{y=a}

<u>PushTopAxiom-push.haha</u>

```
function S() : Z
  var
    sindex : Z
    sarray: ARRAY[Z]
    y : Z
    a : Z
    error: Z
  begin
  skip
```

```
  {true}
  sindex:=0
  {sindex = 0}
  sindex:=sindex+1
  {sindex = 1}
  sarray[sindex]:=a
  {sindex = 1 /\ sarray[sindex]=a }
  sindex:=sindex+1
  {sindex = 2}
  sarray[sindex]:=a
  {sindex = 2 /\ sarray[sindex]=a }
  if (sindex>0) then
     y := sarray[sindex]
  else
     y:= error
  { y = a }
  skip
end
```

HAHA Result:

```
Total number of conditions: 8
Valid: 8
Invalid: 0
Unknown: 0
Errors: 0
Missing: 0
```

3. **Push Top Axiom(pop):** *v: {**true**} init(); pop(); push(a); y=top() {y=a}*

   v: {true}

sindex:=0;

if (sindex>0) then y:= sarray[sindex-1]

sindex:=sindex+1; sarray[sindex]:=a;

if (sindex>0) {y := sarray[sindex];} else {y:= error;}

{y=a}

PushTopAxiom-pop.haha

```
function S() : Z
  var
    sindex : Z
    sarray: ARRAY[Z]
    y : Z
    a : Z
    error: Z
  begin
```

```
  skip
  {true}
  sindex:=0
  {sindex = 0}
  if (sindex>0) then
  y:= sarray[sindex-1]
  {sindex = 0 }
  sindex:=sindex+1
  {sindex = 1 }
  sarray[sindex]:=a
  {sindex = 1 /\ sarray[sindex]=a}
  if (sindex>0) then
     y := sarray[sindex]
  else
     y:= error
  { y = a }
  skip
end
```

HAHA Result:

```
Total number of conditions: 8
Valid: 8
Invalid: 0
Unknown: 0
Errors: 0
Missing: 0
```

Another example on verifying ADT's implementation using Hoare Logic can be found in APPENDIX-E. The example shows how to verify Queue Data Type starting from axioms of Queue specification ending with automated verification using HAHA tool.

SUMMARY

This chapter presents the advocated approach for verifying ADT implementations against the axioms of the specification, using Hoare logic. It discusses three premises of the approach in detail. These premises are:

- Writing the specification of Abstract Data Types in a behavioral style using a trace-like notation.
- Defining the specifications inductively by means of axioms and rules.
- Verifying implementations against axioms of the specifications using Hoare's logic.

As known, what makes Hoare logic difficult to apply on a large scale, and in particular, what makes it difficult to automate is the need to invent invariant assertions for iterative constructs in programs. The chapter discusses how the taken approach obviates these obstacles associated with Hoare logic. It also gives some examples of how to verify ADT implementations using this approach. Finally, it concludes with examples for verifying ADT implementations automatically using HAHA tool.

CHAPTER EIGHT
IMPLEMENTATION AND DEPLOYMENT

8.1 INTRODUCTION

Since the beginning of software engineering, the world experienced a diversified discussion on the related advantages of software testing against static program analysis including effectiveness, scalability, ease of use, etc. The effective position of each of these techniques is taken against some types of specifications and less effective against other types. Also, very frequently one technique or another is found difficult to be used. This difficult use is not resulted from any intrinsic shortcoming of the technique, but it resulted from the fact that the technique is applied against the wrong kind of specification [7].

Obviously, people do not always choose the specification against which they should ensure product correctness. However, they can actually break down a complex specification into smaller components and map each component to the most adaptable technique. The specification of a state-based software product is considered in the form of axioms and rules, and a candidate implementation is considered in the form of a class (i.e. an encapsulated module that maintains an internal state and allows access to a number of externally accessible methods). In order to verify the correctness of the proposed implementation with respect to the specification (i.e. verify the implementation against the axioms), one resolves to proceed as follows:

Each axiom of the specification can be mapped onto a Hoare formula. Such an axiom has a *True* pre-condition and its post-condition is a statement about the behavior specified by the axiom. In the notations introduced in **Chapter 3**, axioms have the form: R(h) =y where *h* is an (elementary) input history that ends with a *VX* symbol (the symbol represents a method that returns a value but does not change the state) and *y* is the corresponding output. History *h* can then be written as h= *h* vop, where *vop* is a *VX* symbol. The following axiom is considered in the specification of the stack ADT: stack(init.push(a).top)=a, and *g* is let be a candidate implementation with a method of the same name as each input symbol of the specification. Then to verify the correctness of the implementation the following formula is generated:

v: {*true*} **g.init(); g.push(a); y=g.top();** {y = a}

where *y* is a variable of type itemtype and *a* is a constant of the same type. By definition, the above formulas in fact deal with non-significant cases. Therefore, they do not involve the issues that usually make correctness proofs complicated. Particularly, they typically involve simple intermediate assertions and do not involve complex invariant assertion generation.

The present chapter presents the proposed verification model for verifying the correctness of ADT implementations against the axioms of specifications and shows how this model is implemented and deployed [97].

8.2 THE FRAMEWORK OF THE VERIFICATION MODEL

The proposed verification model is shown in Figure 8.1, which is a black box diagram that has two inputs (ADT specification and ADT implementation) and one output (verification report) [97].

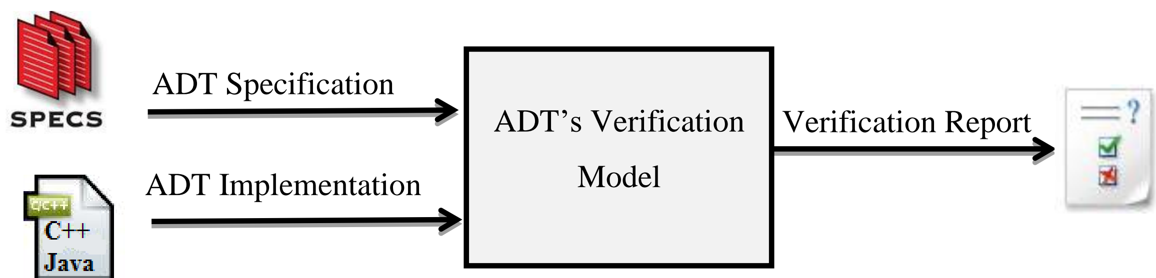


Figure 8.1: The Verification Model

Figure 8.2 shows the framework for the verification model in which a detailed description of the verification model is presented. The steps of *Alneelain* compiler design is discussed in detail in **Chapter 6.** Here the focus is on proof support tool, which is separated into two steps:

1. Mapping axioms into Hoare formulas: Each axiom of the specification can be mapped onto a Hoare formula , such as:

   v: {*true*} **init(); push(a); y=top();** {y = a}

2. Putting Hoare formulas in the syntax of HAHA tool: to verify the correctness of Hoare formulas HAHA tool (discussed in detail at **Chapter 5**) is selected to be used.

Figure 8.3 shows the proof support algorithm. The algorithm presents the steps that are taken to implement the proof support tool:

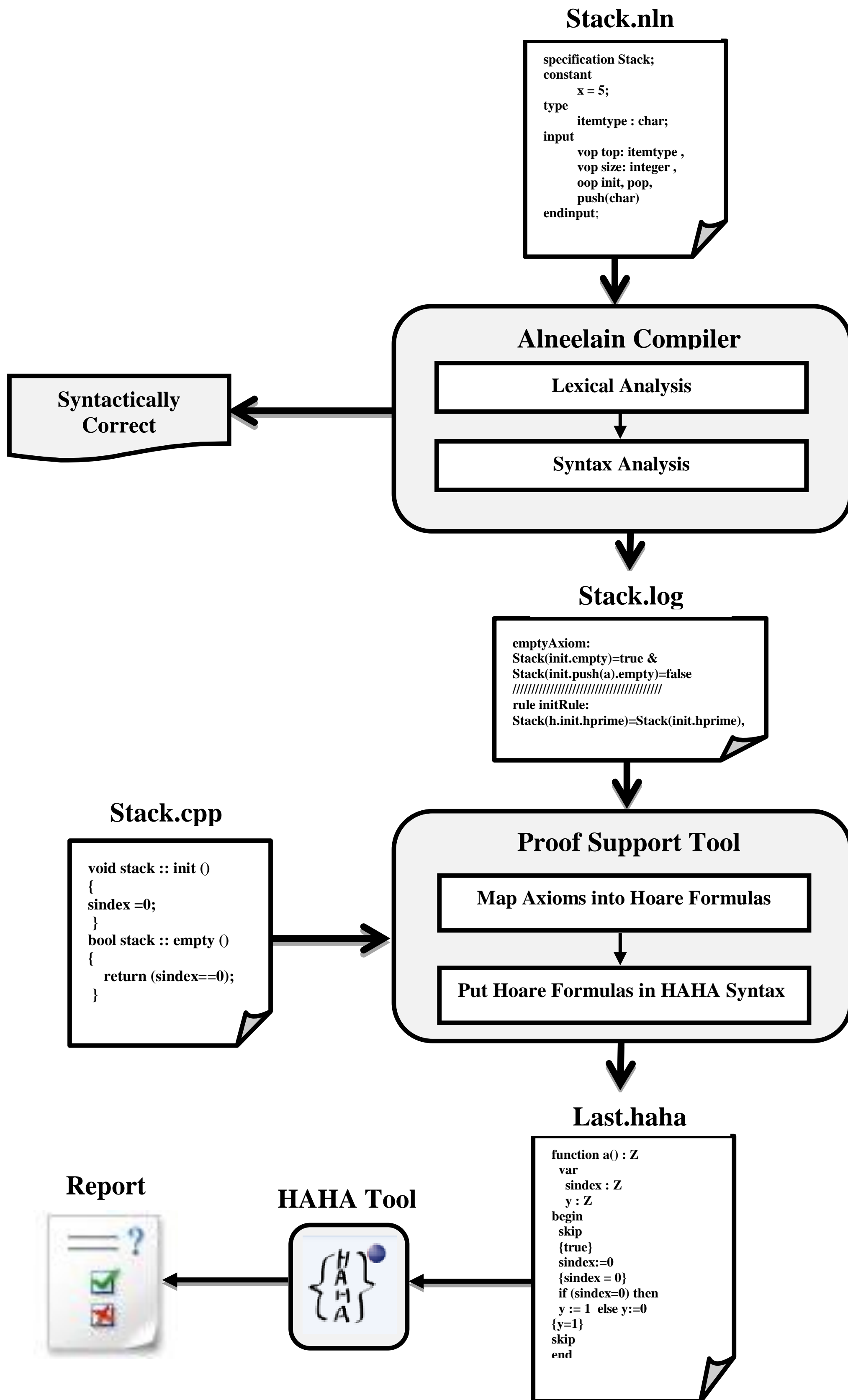


Figure 8.2: The Framework for the Verification Model

Step1: Open Stack.log file.

Step2: Search inside the Stack.log file for axioms.

Step3: Take one of the axioms.

Stack(init.empty) = true

Step4: Map the axiom to Hoare formula to generate the verification condition.

*v: {**true**} init(); y=empty() {y=true}.*

Step5: Open the implementation file (Stack.cpp).

```
void stack :: init ()
{
        sindex =0;
}
bool stack :: empty ()
{
        return (sindex==0);
}
```

Step6: Search inside the implementation file for all the functions included in the axiom, such as init( ); and empty( ); .

Step7: Change each function name in verification condition by its body from the implementation file.

```
v: {true}
                sindex =0;
                 y= (sindex==0);
                                        {y=true}.
```

Step8: Put the verification condition in the syntax of HAHA tool.

```
function a() : Z
var
        sindex : Z
        y : Z
begin
skip
{true}
sindex:=0
{sindex = 0}
if (sindex=0) then
        y := 1  else y:=0
{y=1}
skip
end
```

Step9: Save the produced file with .haha extension.

Step10: Pass the .haha file to the HAHA tool for proving it.

Step11: Run HAHA tool to verify the correctness of the verification condition.

Step12: Store the result.

Step13: Repeat step3 to step 12 until take all the axioms in Stack.log file.

Step14: Interpret the result.

Figure 8.3: Proof Support Algorithm

8.3 SOFTWARE DEPLOYMENT

In order to implement and deploy the proof support tool and connect it with HAHA tool for verification purpose, three main software products are used. As result, a system or application called *Alneelain* verification system is produced [97]. There are many requirements to develop *Alneelain* verification system:

1. An Application software that can be run on various software and hardware platforms.
2. An Attractive and pleasing Graphical User Interface (GUI).
3. A C++ compiler to be used for specification language compiler construction.
4. A tool that helps in searching the windows of HAHA tool in order to interface with it for the purpose of verification.
5. A programming language that is used for writing the code that connect *Alneelain* user interface with HAHA tool.

Those requirements can be achieved by using three products: Qt software can help in developing a wonderful GUI and its support C++ compiler. In addition, it is provided a cross-platform application framework. Spy++ is used for searching and viewing the properties of selected HAHA windows. Visual Studio to help in connecting user interface with HAHA tool. In the subsequent sections, we discuss those software products in detail and present their features.

8.3.1 QT SOFTWARE

Qt is recognized as a cross-platform application framework. It is widely used for developing application software that can be run on various software and hardware platforms. The development of the application can be with little or no change in the underlying codebase [98]. Despite this Qt is still being a useful application with the capabilities and speed thereof.

Qt is primarily used for developing application software characterized with Graphical User Interfaces (GUIs). However, applications without a GUI can also be developed. These include command-line tools and consoles for servers. An example of a non-GUI program developed using Qt is the Cutelyst web framework. GUI programs created with Qt can have a native-looking interface, in which cases Qt is classified as a widget toolkit [98].

Qt usually utilizes standard C++ with extensions including signals and slots that simplify handling of events. This assists in the development of both GUIs and server applications, which receive their own set of event information, and Qt should process them accordingly. Qt supports many compilers, including the GCC C++ compiler and the Visual Studio suite. Qt Quick, a version of Qt, includes a declarative scripting language called QML that allows using JavaScript to provide the logic. Qt Quick enables the possibility of rapid application development for mobile devices, although logic can be written with native code as well to accomplish the best attainable performance. Moreover, Qt can be utilized in numerous programming languages via language bindings. It can also be run on the main desktop platforms and on some mobile platforms. It has a wide global support. Non-GUI features of Qt include SQL database access, XML parsing, JSON parsing, thread management and network support. With Qt for Application Development, it is possible to [98]:

- Create wonderful and intuitive experiences for all of users.
- Code once and then deploy across all screens and platforms.
- Increase development productivity and speed up developer's time to market.
- Simplify and future-proof developer's technology strategy.
- Streamline developer's workflow and reduce costs.

8.3.2 MICROSOFT VISUAL STUDIO

Microsoft Visual Studio is developed by Microsoft Company as an Integrated Development Environment (IDE) [99]. MS Visual Studio is utilized to develop computer programs for Microsoft Windows, in addition to web applications and web services. It utilizes Microsoft software development platforms such as Windows API, Windows Forms, Windows Presentation Foundation, Windows Store and Microsoft Silverlight. It has the ability to produce both native code and managed code. The program includes a code editor that supports IntelliSense (the code completion component) as well as code refactoring. The integrated debugger of MS Visual Studio works as both a source-level debugger and a machine-level debugger. Its other built-in tools comprise several designers: a designer for building GUI applications, web designer, class designer, and database schema designer. It allows plug-ins that enhance the functionality at almost every level. This includes adding support for source-control systems (like Subversion) and adding new toolsets like editors and visual designers for domain-specific languages or toolsets for other aspects of the software development lifecycle (like the Team Foundation Server client: Team Explorer) [99]. Visual Studio supports different programming languages. It allows the code editor and debugger to support (to varying degrees) almost all programming languages, on condition that a language-specific service exists. Its built-in languages include C, C++and C++/CLI (via Visual C++), VB.NET (via Visual Basic .NET), C# (via Visual C#), and F# (as of Visual Studio 2010). Support for other programming languages like Python, Ruby, Node.js, and M among others is provided through language services that are separately installed. It also supports XML/XSLT, HTML/XHTML, JavaScript and CSS. Java. Java (and J#) were supported in the past [99].

8.3.3 SPY++

Spy++ is a Win32-based utility. It gives the user a graphical view of the system's processes, threads, windows, and window messages [100]. Spy++ has a toolbar and hyperlinks to help users work faster. In addition, it provides a Refresh command to update the active view including a Window Finder Tool to make spying easier, and a Font dialog box to customize view windows. Moreover,

Spy++ saves and restores user preferences. Spy++ allows users to perform the following tasks [100]:

- Displaying a graphical tree of relationships among system objects including processes, threads, and windows.
- Searching for specified windows, threads, processes, or messages.
- Viewing the properties of selected windows, threads, processes, or messages.
- Selecting a window, thread, process, or message directly in the view.
- Using the Finder Tool to select a window by mouse pointer positioning.

8.4 USER INTERFACE OF ALNEELAIN VERIFICATION SYSTEM

Alneelain Verification System requires HAHA tool version 0.56, which can be downloaded from http://haha.mimuw.edu.pl/. To run *Alneelain* Verification System, launch the executable file (.exe file) from the installation directory at https://sites.google.com/site/nahidahmedali/. When started, the application will appear as shown below in Figure 8.4.

The most important parts of the application window are the *specification* pane, the *implementation* pane and the *result* pane. The *specification* pane allows user to write a specification in the syntax of *Alneelain* Specification Language and checks if it's syntactically correct or not. The *implementation* pane allows user to write the implementation of an abstract data type. The *result* pane shows the result of verifying the user implementation. Through the application, one can check if the implementation is correct with respect to axioms of specification. *Alneelain* verification system contains five main menus and one toolbar. The toolbar provides instant access to the most important commands from the menu. The five menus are:

1. File Menu:

Through the *File* menu, the user can do the following:

New File: This item prompts the user to create a file with an empty text editor. In the specification pane, the user can write the specification in the syntax of *Alneelain* Specification Language using the keyboard. In implementation pane, the user can write the implementation using the keyboard. The cursor will appear at current line in editor pane.

Open File: This item prompts the user for the name and location of an existing specification or implementation file, and opens it in the application.

Save File: This item allows the user to save the file that is open in the current tab of the editor window to its existing name. Note that the specification file should have the *.nln* extension and the implementation file should have the *.cpp* extension.

Save File As: This item allows the user to save the file that is open in the current tab of the editor window, but under a different name. Note that the specification file should have the *.nln* extension and the implementation file should have the *.cpp* extension.

Exit: This item quits the application. It prompts the user to save any unsaved files that are open in the editor.

2. Edit Menu:

This feature allows a selection of text to be copied or cut. Paste command places the text or object on the clipboard at the current cursor location in the currently active view or editor. Undo command reverses the user's most recent editing action and Redo command re-applies the editing action that has most recently been reversed by the undo action.

3. CheckSpec Menu:

This feature causes the open file in the specification pane to be processed. Checks for syntactic is made, this can be used to check quickly for simple errors. After performing a check, an alert box will appear that shows if the specification is syntactically correct or not. Suggestions and errors messages may also appear.

4. Verify Menu:

This feature allows the user to verify the implementation of an abstract data type with respect to axioms of specification and produce the verification report.

5. Help Menu:

This menu provides help on using the *Alneelain* Verification System. Specifically, the help menu includes user manual and the about command which displays information about the application and installed features.

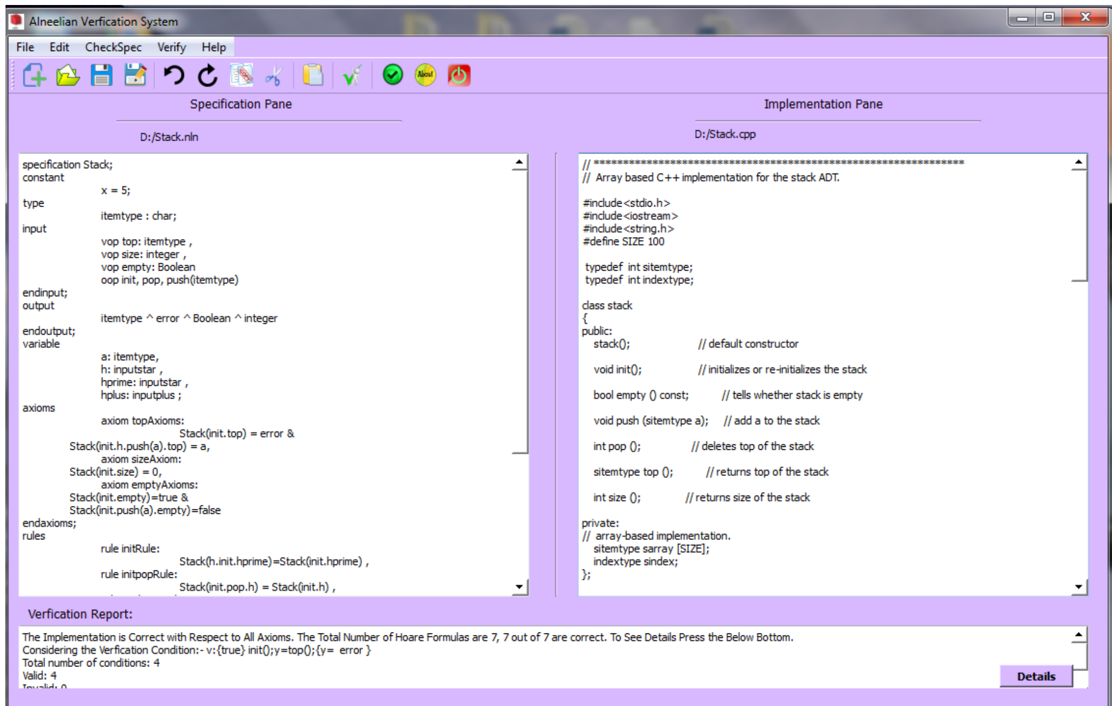


Figure 8.4: The User Interface of Alneelain Verification System

SUMMARY

This chapter presents a proposed verification model for verifying the correctness of ADT implementations against the axioms of specification. A detailed framework for the proposed model is presented. The Algorithm for proof support tool, which is used for mapping the axioms of specification into Hoare formulas and putting these formulas in syntax of HAHA tool in order to verify their correctness, is discussed. In addition, the chapter presents three different software that are used for implementing and deploying the proposed model. Finally, the chapter concludes with a discussion of the user interface of *Alneelain* Verification System, which is a system used for writing an ADT specification and implementation and verify the correctness of the implementation with respect to the specification.

CHAPTER NINE
AUTOMATED VERIFICATION AND VALIDATION

9.1 INTRODUCTION

The verification processes presented in **Chapter 7** are based on the ADT specification and hence can be developed and deployed on a candidate ADT without having to look at the ADT implementation. The verification process can refer to two distinct implementations [7]:

- A traditional implementation based on an array and an index.
- An implementation based on a single integer that store the elements of the stack as successive digits in a numeric representation. The base of the numeric representation is determined by the number of symbols that we wish to store in the stack.

The motivation of having two implementations is to highlight that the verification process does not depend on candidate implementations; the purpose of the second implementation, as counterintuitive as it is, is to highlight the fact that the specifications are behavioral [7]. That means they specify exclusively the externally observable behavior of software systems and make no assumption/prescription on how this behavior ought to be implemented. Also note that the used behavioral specifications do not specify individually the behavior of each method; rather they specify collectively the inter-relationships between these methods, leaving all the necessary latitude to the designer to decide on the representation and the manipulation of the state data. The header files of the two implementations are virtually identical, except for different variable declarations (an array and an index in the first case, a single integer, and a constant base for the second). The ***.cpp*** file for array-based implementation for Stack Data Type is shown in **Chapter 7** at Figure 7.1 and *.cpp* file for the integer-based implementation is written as follows (See Figure 9.1):

```
// ****************************************************
// Scalar based C++ implementation for the stack ADT.
// file stack.cpp, refers to header file stack.h.
// base is declared as a constant in the header file, =8.
// ****************************************************

#include "stack.h"
#include <math.h>

stack :: stack ()
{
}

void stack :: init ()
{
    n=1;
}

bool stack :: empty () const
{
    return (n==1);
}

void stack :: push (itemtype sitem)
{
    n = n*base + sitem;
}

void stack :: pop ()
{
    if (n>1)
    { // stack is not empty
        n = n / base;
    }
}

itemtype stack :: top ()
{
    int error = -9999;
    if (n>1)
    {
        return n % base;
    }
    else
    {
        return error;
    }
}

int stack :: size ()
{
    return (int) (log(n)/log(base));
}
```

Figure 9.1: Integer-Based Implementation of Stack Data Type

9.2 EXAMPLES OF CORRECT IMPLEMENTATIONS THAT SUCCEED

The considered scenario where a candidate program *g* is verified against the axioms of specification *R*, and it is found that the program *behave correctly* with respect to all axioms (As presented in **Chapter 7**). Firstly, one needs to specify precisely what is meant by *behave correctly* (in reference to a program under verification). The adopted interpretation is that whenever the verification system is executed on candidate program, it returns *valid* for the all generated verification conditions and it does not return *invalid* or *unknown* verification conditions in the verification report. As examples, Figure 9.2 shows a correct array-based implementation that succeed for Stack data type and Figure 9.3 shows the verification report for Stack implementation, which appear when one press the *Details* bottom in the *result* pane. From the report, it is clear that all the verification conditions are correct with respect to all axioms of the specification [97].

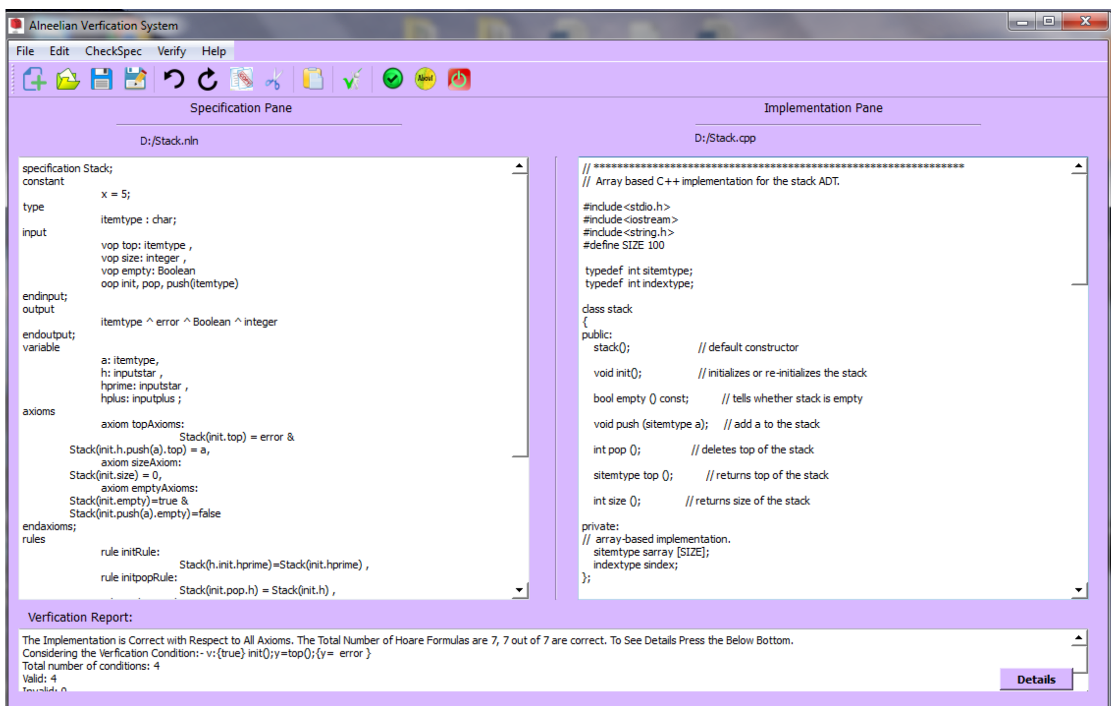


Figure 9.2: An Example for a Correct Array-Based Implementation for Stack

Considering the Verification Condition:- v:{true} init(); y=top(); {y= error}

```
Total number of conditions: 4
Valid: 4
Invalid: 0
Unknown: 0
Errors: 0
```

```
Missing: 0
```

Considering the Verification Condition:- v:{true} init(); init(); push(a);y=top();{y= a}

```
Total number of conditions: 7
Valid: 7
Invalid: 0
Unknown: 0
Errors: 0
Missing: 0
```

Considering the Verification Condition:- v:{true} init(); pop(); push(a);y=top();{y= a}

```
Total number of conditions: 8
Valid: 8
Invalid: 0
Unknown: 0
Errors: 0
Missing: 0
```

Considering the Verification Condition:- v:{true} init(); push(); push(a);y=top();{y= a}

```
Total number of conditions: 8
Valid: 8
Invalid: 0
Unknown: 0
Errors: 0
Missing: 0
```

Considering the Verification Condition:- v:{true} init(); y=size(); {y= 0}

```
Total number of conditions: 3
Valid: 3
Invalid: 0
Unknown: 0
Errors: 0
Missing: 0
```

Considering the Verification Condition:- v:{true} init(); y=empty(); {y=true}

```
Total number of conditions: 4
Valid: 4
Invalid: 0
Unknown: 0
Errors: 0
Missing: 0
```

Considering the Verification Condition:- v:{true} init(); push(a); y=empty();{y=false}

```
Total number of conditions: 6
Valid: 6
Invalid: 0
Unknown: 0
Errors: 0
Missing: 0
```

Figure 9.3: The Verification Report for Stack Implementation

Figure 9.4 shows another example for a correct array-based implementation that succeed for Queue data type and Figure 9.5 shows a correct scalar-based implementation that succeed for Stack data type.

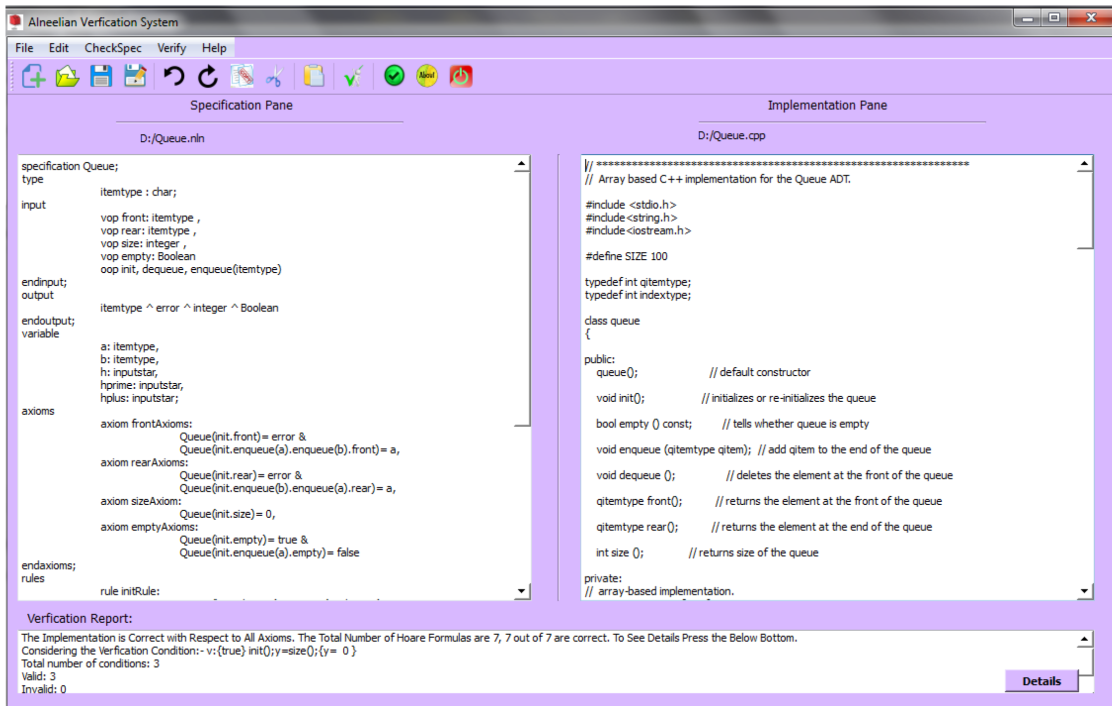


Figure 9.4: An Example for a Correct Array-Based Implementation for Queue

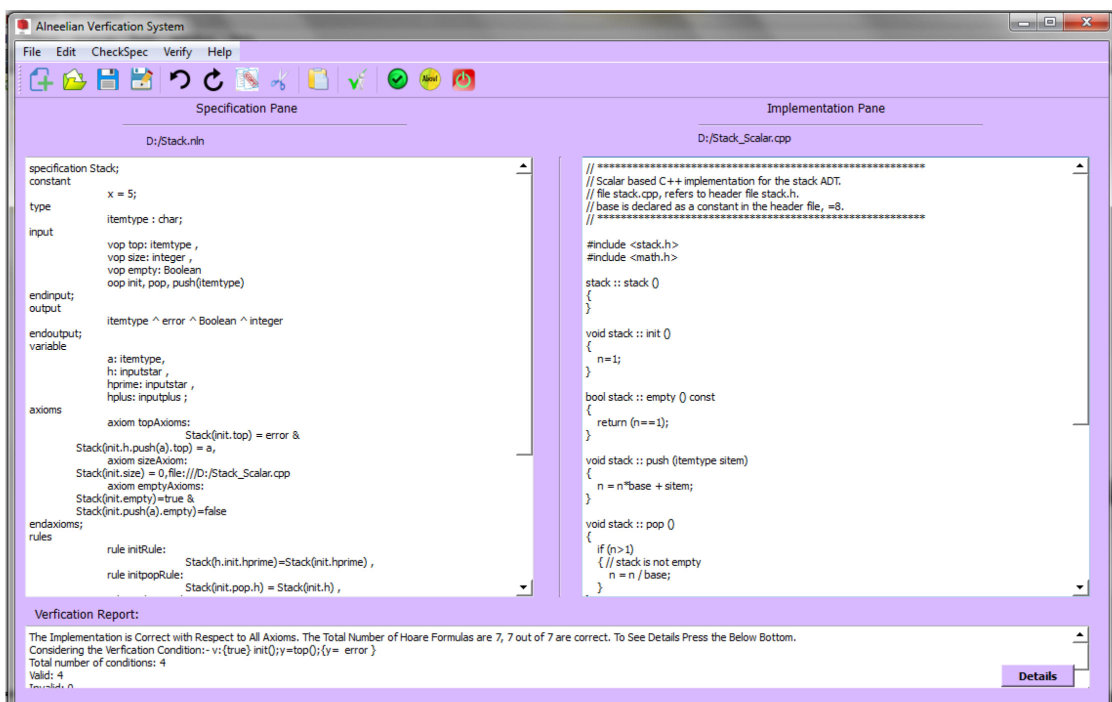


Figure 9.5: An Example for a Correct Scalar-Based Implementation for Stack

9.3 THE ASSESSMENT OF ALNEELAIN VERIFICATION SYSTEM

In order to assess the effectiveness of the developed verification system, we have resolved to introduce faults into the array-based implementation, the scalar-based implementation and the specification, and to observe how the verification system react in terms of proving (or not proving) the correctness of the generated verification conditions [97].

Considering the array-based implementation for Stack and Queue data types, in Table 9.1 and Table 9.2 below some modifications are presented, to which the code is made, and how this affects the output of the verification system is documented.

Table 9.1: Modifications on Array-Based Stack Implementation and their Effects on Verification System

		The Affected Axioms														
		Size axiom			**Init empty axiom**			**Push empty axiom**			**Init top axiom**			**Push top axiom**		
Locus	**Modification**	**Total conditions**	**Valid**	**invalid**	**Total conditions**	**Valid**	**invalid**	**Total conditions**	**Valid**	**invalid**	**Total conditions**	**Valid**	**invalid**	**Total conditions**	**Valid**	**invalid**
pop( );	sindex > 0 → sindex > 1	-	-	-	-	-	-	-	-	-	-	-	-	**8**	**7**	**1**
push( );	sindex++; sarray[sindex]= sitem; → sarray[sindex]= sitem; sindex++;	-	-	-	-	-	-	**6**	**5**	**1**	-	-	-	**7**	**6**	**1**
push( );	sindex++; sarray[sindex]= sitem; sindex++;	-	-	-	-	-	-	**7**	**6**	**1**	-	-	-	**8**	**7**	**1**

Table 9.2: Modifications on Array-Based Queue Implementation and their Effects on Verification System

Locus	Modification	The Affected Axioms														
		Size Axiom			Init Empty Axiom			Enqueue Empty Axiom			Init Front Axiom			Enqueue Front Axiom		
		Total conditions	Valid	invalid	Total conditions	Valid	invalid	Total conditions	Valid	invalid	Total conditions	Valid	invalid	Total conditions	Valid	invalid
init();	qindex =0 front1=-1 rear1=-1 → qindex =1 front1=0 rear1=0	**5**	**4**	**1**	**6**	**5**	**1**	**13**	**12**	**1**	**6**	**5**	**1**	**22**	**21**	**1**
Enqueue();	if(fron1= -1) → if(front1=0)	**-**	**-**	**-**	**-**	**-**	**-**	**12**	**11**	**2**	**-**	**-**	**-**	**22**	**20**	**2**
front();	if(fron1 >= 0) → if(fron1 >=-1)	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**6**	**5**	**1**	**22**	**21**	**1**

For the scalar-based implementation, the founded results are shown in Table 9.3 below:

Table 9.3: Modifications on Scalar-Based Implementation and their Effects on Verification System

Locus	Modification	The Affected Axioms														
		Size axiom			Init empty axiom			Push empty axiom			Init top axiom			Push top axiom		
		Total conditions	Valid	invalid	Total conditions	Valid	invalid	Total conditions	Valid	invalid	Total conditions	Valid	invalid	Total conditions	Valid	invalid
pop( );	n>1 → n>=1	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**9**	**9**	**0**
init( );	n=1 → n=0	**3**	**2**	**1**	**4**	**3**	**1**	**7**	**7**	**0**	**4**	**4**	**0**	**8**	**8**	**0**
push( );	n=n*base+sitem → n= n + base * sitem	**-**	**-**	**-**	**-**	**-**	**-**	**7**	**7**	**0**	**-**	**-**	**-**	**8**	**8**	**0**

From the discussion above its clear, that presenting modifications to ADT's implementation affects the output of the verification system and make many verification conditions to be *invalid*. To continue the assessment of the effectiveness of the verification system, faults are introduced into the specification (Axioms of the specification) of Stack and Queue data types. Table 9.4 and Table 9.5 present the reaction of the verification system in terms of proving (or not proving) the correctness of the generated verification conditions. It is important to note that, this is not the natural way to proceed. Because normally you have more confidence in the specification than in the implementation (program).

Table 9.4: Modifications on Axioms of Stack Data Type and their Effects on Verification System

		The Affected Axioms														
		Size Axiom			**Init Empty Axiom**			**Push Empty Axiom**			**Init Top Axiom**			**Push Top Axiom**		
Locus	**Modification**	**Total conditions**	**Valid**	**invalid**	**Total conditions**	**Valid**	**invalid**	**Total conditions**	**Valid**	**invalid**	**Total conditions**	**Valid**	**invalid**	**Total conditions**	**Valid**	**invalid**
Size Axiom	init.size= 0 → init.size= 1	**3**	**2**	**1**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**
Init Empty Axiom	init.empty=true → init.empty=false	**-**	**-**	**-**	**4**	**3**	**1**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**
Push Empty Axiom	init.push(a). empty = false→ init.push(a). empty = true	**-**	**-**	**-**	**-**	**-**	**-**	**6**	**5**	**1**	**-**	**-**	**-**	**-**	**-**	**-**

Table 9.5: Modifications on Axioms of Queue Data Type and their Effects on Verification System

Locus	Modification	The Affected Axioms														
		Size Axiom			Init Empty Axiom			Enqueue Empty Axiom			Enqueue Front Axiom			Enqueue Rear Axiom		
		Total conditions	Valid	invalid	Total conditions	Valid	Invalid	Total conditions	Valid	invalid	Total conditions	Valid	invalid	Total conditions	Valid	invalid
Size Axiom	Queue(init.size) = 0→ Queue(init.size) = 1	**5**	**4**	**1**	-	-	-	-	-	-	-	-	-	-	-	-
Init Empty Axiom	Queue(init.empty) = true → Queue(init.empty) = false	-	-	-	**6**	**5**	**1**	-	-	-	-	-	-	-	-	-
Enqueue Empty Axiom	Queue(init.enqueue(a). empty) = false→ Queue(init.enqueue(a). empty) = true	-	-	-	-	-	-	**13**	**12**	**1**	-	-	-	-	-	-
Enqueue Front Axiom	Queue(init.enqueue(a). enqueue(b).front) = a → Queue(init.enqueue(a). enqueue(b).front) = b	-	-	-	-	-	-	-	-	-	**22**	**21**	**1**	-	-	-
Enqueue Rear Axiom	Queue(init.enqueue(b) .enqueue(a).rear) = a → Queue(init.enqueue(b) .enqueue(a).rear) = b	-	-	-	-	-	-	-	-	-	-	-	-	**22**	**21**	**1**

SUMMARY

This chapter represents the assessment process of *Alneelain* Verification System. In order to assess the effectiveness of the verification system, we have resolved to introduce faults into the array-based implementation, the scalar-based implementation and the specification, and to observe how the verification system react in terms of proving (or not proving) the correctness of the generated verification conditions. The motivation of having two implementations is to highlight that the verification process does not depend on candidate implementations; the purpose of the scalar-based implementation is to highlight the fact that the specifications are behavioral, that is, they specify exclusively the externally observable behavior of software systems, and make no assumption on how this behavior ought to be

implemented. The chapter also presents some examples of correct implementations that succeed or that *behave correctly* for some ADT's.

CHAPTER TEN
THE INTEGRATION

10.1 INTRODUCTION

Testing is the process of executing a software product on sample input data and analyzing its output [7]. The focus of software testing is to run the candidate program on selected input data and check whether the program behaves correctly with respect to its specification.

It is customary to argue that dynamic testing and static verification are complementary techniques to ensure the correctness or reliability of software products. But, complementarity is meaningful only if the results of these two techniques can be expressed in the same broad framework; specifications make this possible [7].

In the perennial debate about the comparative merits of static analysis and dynamic testing, an important detail often gets overlooked: the observation that what makes a method ineffective is not any intrinsic shortcoming of the method but rather the fact that it is used against the wrong specification. A cost-effective approach to software quality may be to use testing for some parts of the specification and use static analysis for other parts. This approach is possible only when the two methods are designed to deal with the same specification framework. Hence by doing this, we achieve great gains in efficiency, quality, and reliability [7].

It is best to view software testing, not as an isolated effort, but rather as an integral part of a broad, multi-pronged policy of quality assurance that deploys each method where it is most effective by virtue of the *Law of Diminishing Returns* [7].

In this research, we argue in favor of a hybrid approach, whereby testing and proving are used in concert, each being deployed where it works best. Specifically, we envision employing our hybrid proving/ testing approach in the context of verifying the implementation of abstract data types (or other systems that maintain an internal state) that are specified using axiomatic specification (in form of axioms and rules). Whereas verification techniques are used to verify the implementation against the axioms (as discussed in the previous chapters), testing is used to check the implementation against oracles defined by the rules [36]. The integrated verification

and testing tool can be used as an educational tool in an integrated programming environment to teach students how to define and validate a formal specification, verify and test them against the specifications.

10.2 GOAL ORIENTED TESTING

We argue that software testing should be conducted in a systematic manner, by following a rigorous process that includes the following steps [7]:

1. Defining a precise goal for the test (unit testing, integration testing, acceptance testing, certification testing, reliability testing, regression testing, etc.).
2. Defining precise hypotheses under which the test is conducted (what are we assuming to be correct, what are we checking).
3. Defining an oracle that determines whether any given test was successful.
4. Defining a criterion for test data selection.
5. Defining a criterion for when we can consider that the goal of the testing activity has been achieved.
6. Defining how the test results are to be analyzed.
7. Defining the claim that we can make about the software artifact once the test has completed and its results have been analyzed.

So far, we have used Hoare logic to prove that our implementation is correct with respect to the axioms of the specification (As presented in **Chapter 7**); proving that it is also correct with respect to the rules is very difficult, hence we turn to testing. Rules are typically used to build an inductive argument linking the behavior of the specified product on simple input histories to its behavior on more complex input histories. The vast majority of rules fall into two broad classes: a class that provides that two input histories are equivalent and a class that provides an equation between the values of a VX operation at the end of a complex input history as a function of the value of the VX operation at the end of a simpler input history. Representative examples of these categories of rules for the stack specification include the following:

stack(init.h.push(a).pop.h+) = stack(init.h.h+).

stack(init.h.push(a).size) = 1+stack(init.h.size).

This research focus on the first class, which provides that two input histories are equivalent. We use rules of the specification as oracles against which we test the implementation. Specifically, we develop test drivers for the ADT, involving automatic generation of random test data according to selected usage patterns, and we check at run-time that the properties of the rules are satisfied for an arbitrarily large set of data configurations. A testing policy is based on three decisions: how to generate test data; how to generate/ develop an oracle; and how to analyze test outcomes. We review these questions in turn, below:

1. ***Test Data Generation***. We feel that usage-pattern based random test data generation offers the best tradeoff in terms of effort (test data is generated automatically), thoroughness (we can run millions of tests in a single experiment), and coverage (by generating test data according to the usage pattern, we detect the most egregious faults first, hence we maximize the impact of testing) [7]. The main goal of the test data generator is to generate input histories h and h+ at random.

   Note that because of the form of our specifications, test data takes the form of input histories, where each history is a sequence of method calls.

2. ***Oracle Generation***. We resolve to use rules as oracles; in other words, we check through testing whether our implementation satisfies the rules of the axiomatic specification. Many of these rules involve sequences of method calls (h, h', h+, etc.); these are generated randomly, as we discuss above. Also, many of the rules express an equivalence between module states, which raises the question: what does it mean for two states to be equivalent. We adopt the following answer: We consider that two states *s* and *s'* are equivalent (for the purposes of our test) if each VX operation returns the same value in state *s* and in state *s'*. For example, we want to check that VX operation of Stack (top, size, and empty) return the same value:

   init.h.push(a).pop.h+.top = init.h.h+.top

   init.h.push(a).pop.h+.size = init.h.h+.size

   init.h.push(a).pop.h+.empty = init.h.h+.empty

   To check this, we first check the test driver. The test driver queries the user about the number of random tests she/he wants to run, as well as the type of error report she/he wants to get, and it produces a test report accordingly.

3. ***How to Analyze Test Data***. For the purposes of this phase, we are only interested to record the failure rate of the module when it is submitted to a large set of test data. We are not removing any faults in this phase, only recording how often the module execution satisfies the oracle, and how often it fails to. In a way, this phase is not testing our ADT code as much as it is testing the test driver: it is really ensuring that our test driver is reliable.

In light of these decisions, the outermost structure of our test driver look as shown in Figure 10.1 below:

```
#include <iostream>
#include "stack.cpp"
#include "rand.cpp"
using namespace std;
typedef int boolean;
typedef int itemtype;
const int testsize = 10000;
const int hlength = 9;
const int Xsize = 5;
const itemtype paramrange=7; // drawing parameter to push()
// random number generators
int randnat(int rangemax); int gt0randnat(int rangemax);
// rule testers
void initrule(); void initpoprule(); void pushpoprule();
void sizerule(); void emptyrulea(); void emptyruleb();
void vopruletop(); void voprulesize(); void vopruleempty();
// history generator
void historygenerator
(int hl, int hop[hlength], itemtype hparam[hlength]);
/* State Variables */
stack s; // test object
int nbf; // number of failures
int main ()
{
/* initialization */
```

```
nbf=0; // counting the number of failures
SetSeed (825); // random number generator
for (int i=0; i<testsize; i++)
{switch(i%9)
{case 0: initrule(); break;
case 1: initpoprule(); break;
case 2: pushpoprule(); break;
case 3: sizerule(); break;
case 4: emptyrulea(); break;
case 5: emptyruleb(); break;
case 6: vopruletop(); break;
case 7: voprulesize(); break;
case 8: vopruleempty(); break;}
}
cout << "failure rate: " << nbf << " out of " << testsize
<< endl;
}
```

Figure 10.1: The Test Driver

This loop will cycle through the rules, testing them one by one successively. The factor **testsize** determines the overall size of the test data; because test data is generated automatically, this constant can be arbitrarily large, affording us an arbitrarily thorough test. The variable **nbf** represents the number of failing tests, and is incremented by the routines that are invoked in the switch statement, whenever a test fails. For the sake of illustration, we consider the function pushpoprule(), which we detail below as shown in Figure 10.2:

```
void pushpoprule()
{
// stack(init.h.push(a).pop.h+) = stack(init.h.h+)
int hl, hop[hlength]; itemtype hparam[hlength];
// storing h
int hplusl, hplusop[hlength]; itemtype hplusparam
[hlength]; // storing h+
```

```
int storesize; // size in LHS
boolean storeempty; // empty in LHS
itemtype storetop; // top in LHS
boolean successfultest; // successful test
// drawing h and h+ at random, storing them in hop, hplusop
hl = randnat(hlength);
for (int k=0; k<hl-1; k++)
{hop[k]=gt0randnat(Xsize);
if (hop[k]==1) {hparam[k]=randnat(paramrange);}}
hplusl = gt0randnat(hlength);
for (int k=0; k<hplusl-1; k++)
{hplusop[k]=gt0randnat(Xsize);
if (hplusop[k]==1) {hplusparam[k]=randnat
(paramrange);}}
// left hand side of rule
s.sinit(); historygenerator(hl,hop,hparam);
itemtype a=randnat(paramrange); s.push(a); s.pop();
historygenerator(hplusl,hplusop,hplusparam);
// store resulting state
storesize = s.size(); storeempty=s.empty();
storetop=s.top();
// right hand side of rule
s.sinit(); historygenerator(hl,hop,hparam);
historygenerator(hplusl,hplusop,hplusparam);
// compare current state with stored state
successfultest =
(storesize==s.size()) && (storeempty==s.empty()) &&
(storetop==s.top());
if (! successfultest) {nbf++;}
}
```

Figure 10.2: The Source Code of Pushpoprule() Function

We have included comments in the code to explain it. Basically, this function proceeds as follows: First, it generates histories *h* and *h+*; then it executes the sequence *init.h.push(a).pop.h+*, for some arbitrary item *a*; then it takes a snapshot of

the current state by calling all the VX operations and storing the values they return. Then it reinitializes the stack and calls the sequence *init.h.h+;* finally, it verifies that the current state of the stack (as defined by the values returned by the VX operations) is identical to the state of the stack at the sequence given in the left hand side (which was previously stored). If the values are identical, then we declare a successful test; if not, we increment **nbf**.

Once we generate a function for each of the six rules, we can run the test driver with an arbitrary value of variable testsize (to make the test arbitrarily thorough), an arbitrary value of variable hlength (to make h sequences arbitrarily large), and an arbitrary value of variable paramrange (to let the items stored on the stack take their values from a wide range).

Execution of the test driver on our stack with the following parameter values

- testsize = 10;
- hlength = 5;

yields the following outcome:

failure rate: 0 out of 60

which means that all 60 executions of the stack were consistent with the rules of the stack specification. Of course, typically, when we are dealing with a large and complex module, a more likely outcome is to observe a number of failures. Notice that because test data generation and oracle design are both based on an analysis of the specification, we have written the test driver without looking at the candidate implementation; this means, in particular, that this test driver can be deployed on any implementation of the stack that purports to satisfy the specification we are using.

10.3 THE USER INTERFACE OF THE INTEGRATED TOOL

To run *Alneelain* Verification and Testing System, launch the executable file (.exe file) from the installation directory at https://sites.google.com/site/nahidahmedali/. When started, the application will appear with its three main application windows that are: the *specification* pane, the *implementation* pane and the *result* pane. The *specification* pane allows user to write a specification in the syntax of *Alneelain* Specification Language and checks if it's syntactically correct or not. The *implementation* pane allows user to write the implementation of an abstract data type. The *result* pane shows the result of verifying and testing the user

implementation. Once the user clicks on test icon, the testing parameters window will appear to allow user to determine the number of test runs and the max length for input histories (h, hprime, hplus) then user click on ok bottom to proceed as shown in Figure 10.3 below.

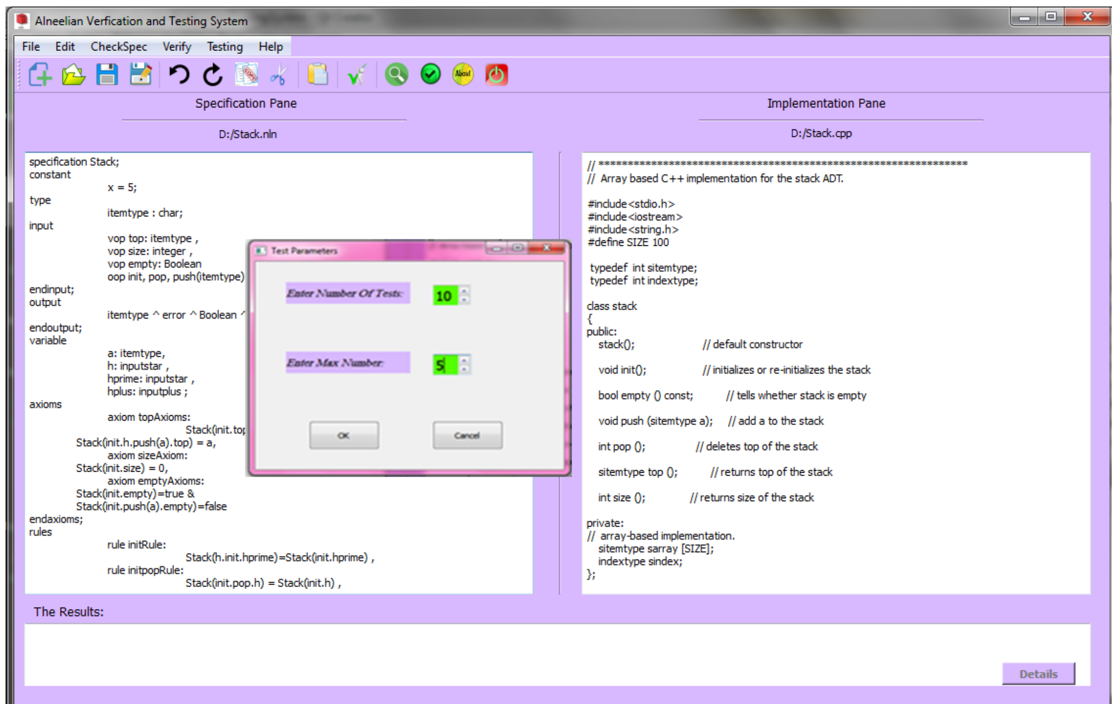


Figure 10.3: The Testing Parameters Window

After that, the test results will appear at result pane which shows the user if the test finished with error or not. For example if the user executes the test driver on the stack data type with the test size 10 and max length for input histories (h, hprime, hplus) 5, the test result will be:

Failure Rate: **0** out of **60,**

as shown in Figure 10.4 below which means that all 60 executions of the stack were consistent with the rules of the stack specification.

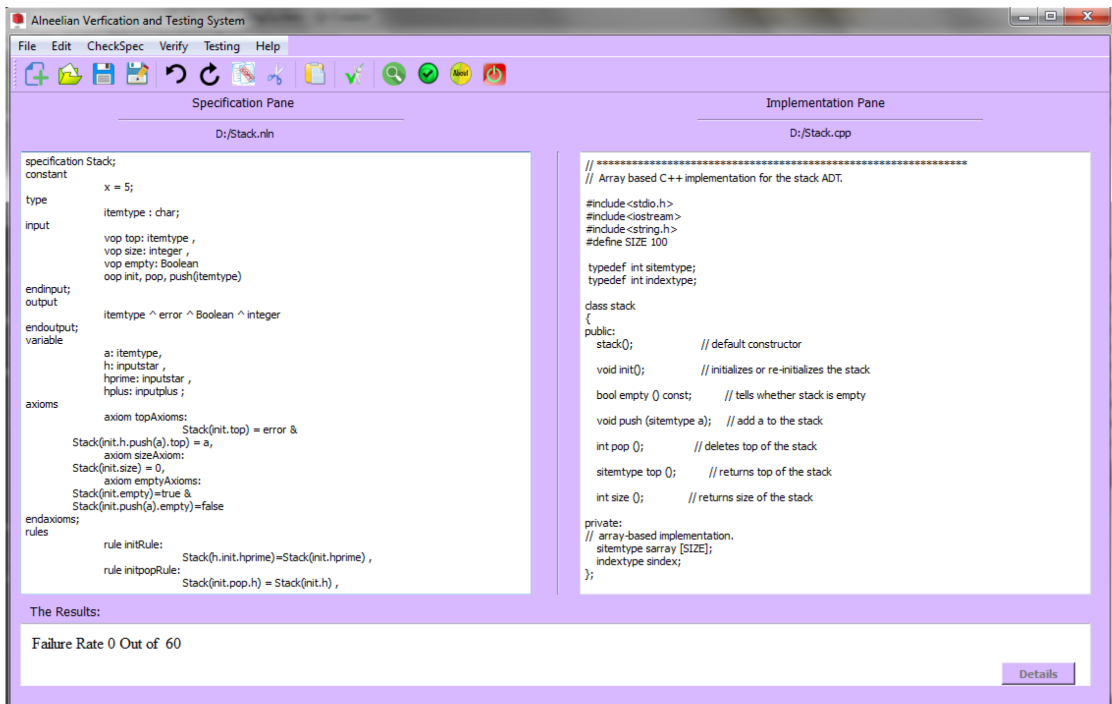


Figure 10.4: The Test Results of the Stack Data Type Implementation

10.4 THE ASSESSMENT OF ALNEELAIN VERIFICATION AND TESTING SYSTEM

In order to assess the effectiveness of the test drivers we have developed, we have resolved to introduce faults into the array-based implementation and the scalar-based implementation, and to observe how the test drivers react in terms of detecting (or not detecting) failure.

Considering the array-based implementation, we present at Table 10.1 below some modifications we have made to the code, and document how this affects the performance of the test drivers and to observe how the test drivers react in terms of detecting (or not detecting) failure.

Table 10.1: Modifications on Array-Based Stack Implementation and their Effects on Verification and Testing System

Locus	Modification	Random test data generation
pop( );	sindex > 0 → sindex > 1	Failure Rate 5 Out of 60
push( );	sarray [sindex] = sitem ; sindex - - ;	Failure Rate 23 Out of 60
push( );	sindex ++ ; sarray [sindex] = sitem ; sindex ++ ;	Failure Rate 1 Out of 60

For the scalar-based implementation, we find the following results as shown in Table 10.2:

Table 10.2: Modifications on Scalar-Based Stack Implementation and their Effects on Verification and Testing System

Locus	Modification	Random test data generation
pop( );	n>1 → n>=1	Failure Rate 4 Out of 60
init( );	n=1 → n=0	Failure Rate 21 Out of 60
push( );	n=n*base+sitem → n=n+base*sitem	Failure Rate 2 Out of 60

SUMMARY

Software testing is the process that focus on running the candidate program on selected input data and check whether the program behaves correctly with respect to its specification. The dynamic testing can be considered as complementary techniques to static verification to ensure the correctness or reliability of software products. This chapter discusses the integration of the verification system discussed in the previous chapters with a testing system (this research is part of a broader project, which also includes a testing component [36] and a validation component [101]. This system can be used as an educational tool to teach students how to define and validate a formal specification, verify and test them against the specifications. We envision employing

our hybrid proving/ testing approach in the context of verifying the implementation of abstract data types that are specified using axiomatic specification. Whereas verification techniques are used to verify the implementation against the axioms, testing is used to check the implementation against oracles defined by the rules. The chapter also discussed the testing policy that is used in this research which based on three decisions: how to generate test data; how to generate/ develop an oracle; and how to analyze test outcomes. In addition, the chapter represents the assessment process of the verification and testing system. In order to assess the effectiveness of the system, we have resolved to introduce faults into the array-based implementation and the scalar-based implementation, and to observe how the system react in terms of detecting (or not detecting) failure.

CHAPTER ELEVEN

CONCLUSION AND FUTURE WORK

11.1 CONCLUSION

This research focuses on program verification as a part of a team effort that involves specification generation and validation, program verification, and testing. The specification of a state-based software product in the form of axioms and rules is considered in this research and a new specification language called *Alneelain* is designed based on axiomatic specification.

In order to verifying ADT implementations against the axioms of the specification, the present study considered a candidate implementation in the form of a class (an encapsulated module that maintains an internal state and allows access to a number of externally accessible methods) in an object-oriented language. Implementations are verified against axioms of the specifications using Hoare's logic; because axioms represent only part of the ADT behavior, verifying implementations against axioms is not sufficient. As a supplement, one can test implementations according to the cleanroom testing discipline, using the rules of the specification as oracles.

As everyone knows, what makes Hoare logic difficult to apply on a large scale, and in particular, what makes it difficult to automate is the need to invent invariant assertions for iterative constructs in programs. However, our approach obviates this obstacle because axioms represent simple basic behavior of the ADT, they do not invoke complex calculations in the source code; at most, they may invoke some loops to initialize a data structure. In this research, two main tools for automatic formal verification of software based on Hoare logic are surveyed. The study focuses on tools that provide some form of formal guarantee, and thus, aid to improve software quality. A short tutorial on these tools is provided, highlighting their differences when applied to practical problems. The evaluation result of tools shows that HAHA is a better choice than KeY-Hoare tool because it offers a high-level language, which can be learned within a short time. Also, the high degree of proof automation together with the ability to provide feedback on failed proof attempts make users comfortable when using it to prove the correctness of their programs. HAHA also removes the difficulty associated with the process of applying Hoare logic manually. The study developed

and deployed a verification system called *Alneelain* Verification System*, which* uses HAHA tool as support tool to automatically verify the correctness of ADT implementations. Through the user interface of *Alneelain* Verification System, users can specify the behavior of ADT's and verify their implementations correctness with respect to the specification. The system is evaluated with several ADT's such as Stack, Queue, Set, etc. The evaluation results show that the system provides a high degree of proof automation combined with the ability to provide feedback on failed proof attempts and thus removes the difficulty associated with the process of applying Hoare logic manually.

11.2 FUTURE WORK

As future prospects, this work can be extended by integration with the validation component. This component is used to validate the ADT's specification written in the syntax of Alneelain specification language. Another extension should be the derivation of an integrated software engineering environment (include verification component, testing component and validation component) that can be used in the classroom for the purpose of supporting courses such as data structures courses.

REFERENCES

[1] Mili , Ali ; Tchier, Fairouz ;, Software Testing: Concepts and Operations, Hoboken, New Jersey: John Wiley & Sons, 2015.

[2] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM,* vol. 12, no. 10, pp. 576 - 580, 1969.

[3] D. Hoffman and R. Snodgrass, "Trace Specifications: Methodology and Models," *IEEE Transactions on Software Engineering,* vol. 14, no. 9, pp. 1243-1252, 1988.

[4] F. Tchier and A. Mili, "On the Verification and Validation of Software Modules : Applications in Teaching and Practice," Tech. Rep. of NJIT, 2013.

[5] IEEE, "IEEE Std 610.12-1990 IEEE Standard Glossary of Software Engineering Terminology," *IEEE Software Engineering Standards Collection ,The Institute of Electrical and Electronics Engineers,New York,* 1990.

[6] D. Galin, Software Quality Assurance: From Theory to Implementation, Pearson education, 2004.

[7] A. Mili and F. Tchier, Software Testing: Concepts and Operations, Hoboken, New Jersey: John Wiley & Sons, 2015.

[8] M. Tedre, The Science of Computing: Shaping a Discipline, CRC Press, 2014.

[9] J. Tian, Software Quality Engineering : Testing, Quality Assurance, and Quantifiable Improvement, Hoboken New Jersey: John Wiley & Sons, 2005.

[10] P. B. Crosby, Quality is Free, New York: McGraw-Hill, 1979.

[11] J. M. Juran and A. B. Godfrey, Juran’s Quality Handbook, 5 ed., McGraw-Hill, 1998.

[12] R. S. Pressman, Software Engineering: A Practioner’s Approach, 5 ed., New York: McGraw-Hill, 2001.

[13] J. Tian, "Quality Assurance Alternatives and Techniques : A Defect-Based Survey and Analysis," *Software Quality Professional,* vol. 3, no. 3, pp. 6-18, 2001.

[14] M. V. Zelkowitz, "Role of Verification in the Software Specification Process," *Advances in Computers,* vol. 36, pp. 43-109, 1993.

[15] E. W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *Communications of the ACM,* vol. 18, no. 8, p. 453–457, 1975.

[16] D. Gries, The Science of Programming, New York Heidelberg Berlin : Springer-Verlag, 1987.

[17] H. D. Mills, V. R. Basili, J. D. Gannon and R. D. Hamlet, Principles of Computer Programming: A Mathematical Approach, Dubuque Iowa: Wm. C. Brown Publisher, 1988.

[18] G. Carlo, J. Mehdi and M. Dino, Fundamentals of Software Engineering, 2 ed., Englewood Cliffs, NJ: Pearson Prentice Hall, 2003.

[19] D. L. Parna and J. Madeyb, "Functional Documents for Computer Systems," *Science of Computer Programming,* vol. 25, no. 1, pp. 41-61, 1995.

[20] N. Wirth, "A Plea for Lean Software," *Computer,* vol. 28, no. 2, pp. 64-68, 1995.

[21] D. L. Parnas, "On the Criteria to be used in Decomposing Systems into Modules," *Communications of the ACM,* vol. 15, no. 12, pp. 1053 - 1058, 1972.

[22] N. Boudriga, A. Mili, R. Zalila and F. Mili, "A Relational Model for the Specification of Data Types," *Computer Languages,* vol. 17, no. 2, pp. 101-131, 1992.

[23] D. L. Parnas, "A Technique for Software Module Specification with Examples," *Communications of the ACM,* vol. 15, no. 5, pp. 330-336, 1972.

[24] B. Liskov and S. Zilles, "An Introduction to Formal Specifications of Data Abstractions," *Current Trends in Programming Methodology,* pp. 1-32, 1977.

[25] B. Meyer, "On Formalism in Specifications," *IEEE Software,* vol. 2, no. 1, pp. 6-26, 1985.

[26] I. Sommerville, Software Engineering, 8 ed., England: Pearson Education Limited, 2007.

[27] V. S. Alagar and K. Periyasamy, Specification of Software Systems, 2 ed., London: Springer-Verlag, 2011.

[28] S. Owre, N. Shankar, J. M. Rushby and D. W. J. Stringer-Calvert, "PVS System Guide," Computer Science Laboratory, SRI International, Version 2.3. Technical Report , Menlo Park, CA, 1999.

[29] D. C. Luckham, F. W. von Henke, B. Krieg-Brueckner and O. Owe, "ANNA: A Language for Annotating ADA Programs (Reference Manual)," vol. 260, no. 0, 1987.

[30] M. Iglewski, M. Kubica, J. Madey, J. Mincer-Daszkiewicz and K. Stencel, "TAM'97: The Trace Assertion Method of Module Interface Specification (Reference Manual)," 1997.

[31] J. Goguen, C. Kirchner, J. Meseguer, H. Kirchner, T. Winkler and A. Megrelis, "An Introduction to OBJ 3," in *1st International Workshop on Conditional Term Rewriting Systems*, London, UK, 1988.

[32] M. J. Spivey, The Z Notation—A Reference Manual, 2 ed., UK: Prentice Hall International, 1992.

[33] J.-R. Abrial, The B-Book—Assigning Programs to Meanings, New York: Cambridge University Press, 1996.

[34] "VDM portal," 10 2015. [Online]. Available: http://overturetool.org/method/.

[35] M. Frentiu , "Correctness: A Very Important Quality Factor in Programming," *Studia Universitas, Babes-Bolyai, Informatica,* vol. L, no. 1, pp. 11-20, 2005.

[36] A. A. M. Yassin, "Testing Abstract Data Types: A Formal Goal Oriented Approach," *International Journal of Engineering Sciences Paradigms and Researches (IJESPR),* vol. 23, no. 01, pp. 18-22, 2015.

[37] A. B. Tucker, Programming Languages, Tata McGraw-Hill Education, 1986.

[38] R. W. Floyd, "Assigning Meanings to Programs," *Mathematical Aspects of Computer Science,* vol. 19, no. 1, pp. 19-32, 1967.

[39] C. A. R. Hoare, "Proof of a Program: FIND," *Communications of the ACM,* vol. 14, no. 1, pp. 39-45, 1971.

[40] M. Foley and C. A. R. Hoare, "Proof of a Recursive Program: Quicksort," *The Computer Journal,* vol. 14, no. 4, pp. 391- 395, 1971.

[41] R. L. London, "Proof of Algorithms: A New Kind of Certification," *Communications of the ACM,* vol. 13, no. 6, pp. 371-373, 1970.

[42] R. L. London, "Proving Programs Correctness: Some Techniques and Examples," *BIT,* vol. 10, pp. 168-182, 1970.

[43] P. Naur, "Proof of Algorithms by General Snapshots," *BIT Numerical Mathematics,* vol. 6, no. 4, pp. 310-316, 1966.

[44] D. Gries, The Science of Programming, New York Heidelberg Berlin: Springer-Verlag, 1981.

[45] G. Dromey, Program Derivation: The Development of Programs from Specifications, Addison Wesley, 1989.

[46] C. Morgan, Programming from Specifications, Prentice Hall International, 1990.

[47] D. Stevenson, "1001 Reasons for Not Proving Program Correct: A Survey," in *Computer Science Dept, Clemson University, steve@ wayne. cs. clemson. edu*, Citeseer, 1990.

[48] H. Leung, "Program Correctness," *AMC,* vol. 10, p. 12, 2010.

[49] J. Loeckx and K. Sieber, The Foundations of Program Verification, Springer Fachmedien Wiesbaden, 1987.

[50] A. Campetelli, "Analysis Techniques: State of the Art in Industry and Research," Tech. Rep., 2010.

[51] E. M. Clarke, J. M. Wing and e. al, "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys,* vol. 28, no. 4, pp. 626 - 643, 1996.

[52] V. D. Silva, D. Kroening and G. Weissenbacher, "A Survey of Automated Techniques for Formal Software Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 27, no. 7, pp. 1165- 1178, 2008.

[53] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages,* pp. 238 - 252, 1977.

[54] P. Cousot and R. Cousot, "Systematic Design of Program Analysis Frameworks," *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages,* pp. 269-282, 1979.

[55] P. Cousot and N. Halbwachs, "Automatic Discovery of Linear Restraints Among Variables of a Program," *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages,* pp. 84-96, 1978.

[56] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival, "Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software," in *The Essence of Computation*, Berlin Heidelberg, Springer, 2002, pp. 85-108.

[57] B. S. Gulavani and S. K. Rajamani, "Counterexample Driven Refinement for Abstract Interpretation," in *Tools and Algorithms for the Construction and Analysis of Systems*, Berlin Heidelberg, Springer, 2006, pp. 474-488.

[58] G. J. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering,* vol. 23, no. 5, pp. 279-295, 1997.

[59] G. J. Holzmann, "Software Model Checking with SPIN," *Advances in Computers,* vol. 65, pp. 77-108, 2005.

[60] W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda, "Model Checking Programs," *Automated Software Engineering,* vol. 10, no. 2, pp. 203- 232, 2003.

[61] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler and D. L. Dill, "CMC: A Pragmatic Approach to Model Checking Real Code," *ACM SIGOPS Operating Systems Review,* vol. 36, no. SI, pp. 75-88, 2002.

[62] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof and Y. Xie, "Zing: Exploiting Program Structure for Model Checking Concurrent Software," in *CONCUR 2004-Concurrency Theory*, Berlin Heidelberg, Springer, 2004, pp. 1-15.

[63] P. Godefroid, "Model Checking for Programming Languages Using VeriSoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, ACM, 1997, pp. 174 - 186.

[64] A. Biere, A. Cimatti, E. Clarke and Y. Zhu, "Symbolic Model Checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, Berlin Heidelberg, Springer, 1999, pp. 193-207.

[65] D. W. Currie, A. J. Hu and S. Rajan, "Automatic Formal Verification of DSP Software," in *Proceedings of the 37th Annual Design Automation Conference*,

ACM, 2000, pp. 130-135.

[66] E. Clarke, D. Kroening and K. Yorav, "Behavioral Consistency of C and Verilog Programs using Bounded Model Checking," in *Design Automation Conference(DAC)*, IEEE, 2003, pp. 368 - 371.

[67] E. Clarke, D. Kroening and F. Lerda, "A Tool for Checking ANSI-C Programs," in *Tools and Algorithms for the Construction and Analysis of Systems(TACAS)*, Berlin Heidelberg, Springer, 2004, pp. 168 - 176.

[68] I. Rabinovitz and O. Grumberg, "Bounded Model Checking of Concurrent Programs," in *Computer Aided Verification*, Berlin Heidelberg, Springer, 2005, pp. 82 - 97.

[69] F. Ivancic, I. Shlyakhter, A. Gupta, M. K. Ganai, V. Kahlon, C. Wang and Z. Yang, "Model Checking C Programs Using F-SOFT," in *International Conference on Computer Design (ICCD)*, IEEE, 2005, pp. 297 - 308.

[70] Y. Xie and A. Aiken, "Scalable Error Detection Using Boolean Satisfiability," *Principles of Programming Languages (POPL),* vol. 40, no. 1, pp. 351 - 363, 2005.

[71] J. Yang, C. Sar, P. Twohey, C. Cadar and D. Engler, "Automatically Generating Malicious Disks Using Symbolic Execution," in *IEEE Symposium on Security and Privacy*, IEEE, 2006, pp. 15 -.

[72] N. A. Ali, "A Survey of Verification Tools Based on Hoare Logic," *International Journal of Software Engineering & Applications (IJSEA),* vol. 8, no. 2, 2017.

[73] T. Nipkow, L. C. Paulson and M. Wenzel, "Isabelle/HOL: A Proof Assistant for Higher-Order Logic," *Springer-Verlag,* 2016.

[74] S. Owre, J. Rushby and N. Shankar, "PVS: A Prototype Verification System," in *11th International Conference on Automated Deduction (CADE)*, vol. 607, Springer-Verlag, 1992, pp. 748-752.

[75] D. Wang, "Symbolic Model Verifier," 1998. [Online]. Available: http://www.cs.cmu.edu/~modelcheck/smv.html.

[76] J. Winkler, "The Frege Program Prover FPP," in *Internationales Wissenschaftliches Kolloquium*, vol. 42, 1997, pp. 116-121.

[77] D. Crocker, "Perfect Developer: A Tool for Object-Oriented Formal Specification and Refinement," *Tools Exhibition Notes at Formal Methods Europe,* 2003.

[78] R. H¨ahnle and R. Bubel, "A Hoare-Style Calculus with Explicit State Updates," *Formal Methods in Computer Science Education(FORMED),* pp. 49-60, 2008.

[79] T. Sznuk and A. Schubert, "Tool Support for Teaching Hoare Logic," in *Software Engineering and Formal Methods*, Springer, 2014, pp. 332-346.

[80] W. Ahrendt, "Key- Hoare System," 2009. [Online]. Available: http://www.key-project.org/download/hoare/.

[81] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337-340.

[82] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi´c, T. King, A. Reynolds and C. Tinelli, "CVC4," in *Computer Aided Verification*, Springer, 2011, pp. 171-177.

[83] T. Pandey and S. Srivastava, "Comparative Analysis of Formal Specification Languages Z, VDM and B," *International Journal of Current Engineering and Technology,* vol. 5, no. 3, pp. 2086-2091, 2015.

[84] R. S. Pressman and B. R. Maxim, Software Engineering: A Practitioner's Approach, 8 Edition ed., New York: McGraw-Hill Education, 2015.

[85] O. M. G. (OMG), "Object Constraint Language," *OMG Available Specification,* vol. Version 2.4, February 2014 (Last Edition).

[86] I. Toyn, "Information Technology- Z Formal Specification Notation -Syntax, Type System and Semantics," *International Standards Origination,* 2002.

[87] J. V. Guttag and J. J. Horning, Larch: Languages and Tools for Formal Specification, Springer-Verlag, 1993.

[88] C. B. Jones, Systematic Software Development using VDM, 2 ed., Prentice Hall International, 1990.

[89] N. A. Ali, A. A. Mirghani and A. Y. Ibrahim, "Alneelain: A Formal Specification Language," in *2017 International Conference on Communication, Control, Computing and Electronics Engineering (ICCCCEE)*, Khartoum,

Sudan, 2017.

[90] P. Naur, J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson and B. Vauquois, "Revised Report on the Algorithmic Language Algol 60," *Communications of the ACM,* vol. 6, no. 1, pp. 1-17, 1963.

[91] K. Slonneger and B. L. Kurtz, Formal Syntax and Semantics of Programming Languages, Addison-Wesley, 1995.

[92] N. Chomsky, "Three Models for the Description of Language," *IRE Transactions on Information Theory,* vol. 2, no. 3, pp. 113-124, 1956.

[93] T. Æ. Mogensen, Introduction to Compiler Design, Springer-Verlag London Limited, 2011.

[94] H. D. Mills, M. Dyer and R. C. Linger, "Cleanroom Software Engineering," *IEEE Software,* vol. 4, no. 5, pp. 19-25, 1987.

[95] N. A. Ali, "Verifying Abstract Data Types: A Hybrid Approach," in *2013 International Conference on Computing, Electrical and Electronic Engineering (ICCEEE)*, Khartoum, Sudan, 2013.

[96] F. Tchier and A. Mili, "On the Verification and Validation of Software Modules: Applications in Teaching and Practice," Tech. Rep. of NJIT, 2013.

[97] N. A. Ali, "Verifying ADT Implementations against Axiomatic Specifications," *International Journal of Mathematics and Statistics Invention (IJMSI),* vol. 5, no. 3, 2017.

[98] J. Blanchette and M. Summerfield, C++ GUI Programming with Qt 4, Prentice Hall, 2006.

[99] Microsoft, "Visual Studio," [Online]. Available: https://www.visualstudio.com/.

[100] Microsoft, "Introducing Spy++," [Online]. Available: https://msdn.microsoft.com/en-us/library/dd460756.aspx.

[101] A. Y. Ibrahim, "Specifying Abstract Data Types:A Behavioral Model, an Axiomatic Representation," in *1st International Conference on Computing, Electrical and Electronic Engineering (ICCEEE 2013)*, Khartoum, Sudan, 2013.

APPENDIX -A

REQUIREMENTS DESCRIPTION FOR ABSTRACT DATA TYPES

Here we present a broad description of the requirements of a number of ADT's namely: Stack, Queue, Sequence, Set, Multiset and List.

1. Stack ADT:

A **stack** is an abstract data type that stores elements in a last in first out (LIFO) order. Elements are added and removed to/from the top only.

O -operations: These are operations that alter the state of the ADT but produce no visible output.

Init: this operation initializes or re-initializes the stack to empty, erasing all past history.

push (itemtype x): this operation pushes an element (provided as parameter) on top of the stack.

pop: this operation removes the top (most recently pushed) element of the stack, if the stack is not empty; else it leaves the stack unchanged.

V-operations: These are operations that return values but do not change the state.

itemtype: top(): returns the top of the stack (last element stored) if the stack is not empty; else it returns an error message.

integer: size(): returns the number of elements of the stack.

boolean: empty(): returns true if and only if the stack is empty.

2. Queue ADT:

A queue is an abstract data type that stores elements in a first-in-first-out order. Elements are added at one end and removed from the other.

O -operations: These are operations that alter the state of the ADT but produce no visible output.

Init: this operation initializes or re-initializes the queue to empty, erasing all past history.

enqueue (itemtype x): this operation adds an element (provided as parameter) at the end of queue.

dequeue: this operation removes the element at the front of the queue, if the queue is not empty; else it leaves the queue unchanged.

V-operations: These are operations that return values but do not change the state.

integer: size(): returns the number of elements of the queue.

boolean: empty(): returns true if and only if the queue is empty.

itemtype: front(): returns the front of the queue (first element stored) if the queue is not empty; else it returns an error message.

itemtype: rear(): returns the rear of the queue (last element stored) if the queue is not empty; else it returns an error message.

3. Sequence ADT:

The sequence data type is one of the fundamental data types in computer science. It consists of a homogeneous ordered collection of objects of any type.

O -operations: These are operations that alter the state of the ADT but produce no visible output.

Init: this operation initializes or re-initializes the sequence to empty, erasing all past history.

puthead (itemtype x): this operation adds an element (provided as parameter) at the head of the sequence.

putlast (itemtype x): this operation adds an element (provided as parameter) at the end of the sequence.

deletehead: this operation removes the element at the head of the sequence, if the sequence is not empty; else it leaves the sequence unchanged.

deletelast: this operation removes the last element of the sequence, if the sequence is not empty; else it leaves the sequence unchanged.

V-operations: These are operations that return values but do not change the state.

integer: length(): returns the number of elements of the sequence.

boolean: empty(): returns true if and only if the sequence is empty.

itemtype: head(): returns the element at the head of the sequence if the sequence is not empty; else it returns an error message.

itemtype: last(): returns the last element in the sequence if the sequence is not empty; else it returns an error message.

4. Set ADT:

A set is an unordered abstract data type that does not allow duplicate elements to be added.

O-operations: These are operations that alter the state of the ADT but produce no visible output.

init: this operation initializes or re-initializes the set to empty.

insert (itemtype x): if x is not in the set, this operation adds it; else it leaves the set unchanged.

remove (itemtype x): if x is not in the set, then this operation is null; else it removes the element x from the set.

V-operations: These are operations that return values but do not change the state.

boolean: empty(): returns true if and only if the set is empty.

integer: size(): returns the number of elements in the set.

boolean: search(itemtype x): tells whether x is in the set.

itemtype: choose(): returns an arbitrary element of the set.

itemtype* list(): lists the elements of the set.

itemtype: smallest(): returns a smallest element if the set is not empty else return plus infinity.

itemtype: largest(): returns a largest element if the set is not empty else return minus infinity.

5. Multiset ADT:

In discrete mathematics, a multiset is a collection of objects where duplication is permitted. We let multiset to be defined by the following operations:

O-operations: These are operations that alter the state of the ADT but produce no visible output.

init: this operation initializes or re-initializes the multiset to empty.

insert (itemtype x): if x is not in the multiset, this operation adds it; if x is in the multiset, it increments its multiplicity.

insert (itemtype x, integer n): inserts n copies of x, where n is non-negative.

remove (itemtype x): if x is not in the multiset, then this operation is null; if x does belong in a single copy, it no longer exists in the set; if x belongs in multiple copies, its multiplicity is reduced by 1.

remove (itemtype x, integer n): performs remove(x) n times.

removeall (itemtype x): if x does not belong to the multiset, then this operation is null; else all instances of x are removed.

removeany(): removes an arbitrary element of the multiset, reducing its multiplicity by 1; if the multiset is empty, this operation is null.

eraseany(): removes all the instances of an arbitrary element; if the multiset is empty, this operation is null.

V-operations: These are operations that return values but do not change the state.

Boolean: empty(): returns true if and only if the multiset is empty.

integer: size(): returns the number of distinct elements.

integer: multisize(): returns the multisize of the multiset.

Boolean: search(itemtype x): tells whether x is in the multiset.

integer: multisearch(itemtype x): gives the multiplicity of x.

itemtype: choose(): returns an arbitrary element of the multiset.

itemtype* list(): lists the elements of the multiset.

itemtype* list(integer n): list the elements of the multiset with a multiplicity greater than or equal to n.

itemtype: least(): returns an element with minimal multiplicity.

itemtype: most(): returns an element with maximal multiplicity.

itemtype: smallest(): returns a smallest element.

itemtype: largest(): returns a largest element.

6. List ADT:

A list ADT stores and retrieves elements in a linearly ordered structure. We let the list be defined by the following operations.

O-operations: These are operations that alter the state of the ADT but produce no visible output.

init: this operation initializes or re-initializes the list to empty.

insertlast (itemtype x): this operation inserts x at the end of the list.

insertfirst (itemtype x): this operation inserts x at the beginning of the list.

insertat (itemtype x, integer n): if the size of the list allows, this operation inserts x at position n; else it does not change the list.

deletelast (): this operation deletes the element at the end of the list.

deletefirst (): this operation deletes the element at the beginning of the list.

deleteat (integer n): if the size of the list allows, this operation deletes the element at position n; else it does not change the list.

V-operations: These are operations that return values but do not change the state.

boolean: empty(): returns true if and only if the list is empty.

integer: size(): returns the number of elements in the list.

boolean: search(itemtype x): tells whether x is in the list.

integer: multisearch(itemtype x): gives the multiplicity of x in the list.

itemtype: choose(): returns an arbitrary element of the list.

itemtype: first(): returns the first element in the list.

itemtype: last(): returns the last element in the list.

itemtype: smallest(): returns a smallest element.

itemtype: largest(): returns a largest element.

APPENDIX -B

THE SPECIFICATIONS OF ABSTRACT DATA TYPES

1. SPECIFICATION OF SEQUENCE

We discuss how to represent the specification of a sequence, in the same way that we wrote the specification of stack and queue. We represent in turn, the input space (from which we infer the set of input histories), then the output space, then the relation, which we denote by sequence.

1. *Input Space*. We let X be defined as:

X = {init, deletehead, deletelast, head, last, length, empty} ∪ {puthead, putlast} × itemtype.

We partition this set into OX = {init, deletehead, deletelast, puthead, putlast} and VX = {head, last, length, empty}. We let H be the set of sequences of elements of X.

2. *Output Space*. We let the output space be defined as:

Y = (itemtype ∪ {error}) ∪ integer ∪ Boolean.

1. ***Axioms***. We propose the following axioms:
   - **Head Axioms**
     - sequence (init.head)= error
       Invoking head on an empty sequence returns an error.
     - sequence (init.h.putop(_)*.puthead(a).head)= a
       Where putop(_)* is any puthead or putlast operation including zero(no operation). Interpretation: Invoking head on a non empty sequence returns the element at the beginning (head) of the sequence.
     - sequence(init.h.puthead(a).putlast(_)*.head)=a
       Where putlast(_)* is any putlast operation including zero. Interpretation: Invoking head on a non empty sequence returns the element at the beginning (head) of the sequence.
   - **Last Axioms**
     - sequence (init.last)= error
       Invoking last on an empty sequence returns an error.
     - sequence (init.h.putop(_)*.putlast(a).last) = a

Invoking last on a non empty sequence returns the element at the end of the sequence.

- sequence(init.h.putlast(a).puthead(_)*.last) = a

  Where puthead(_)* is any puthead operation including zero. Interpretation: Invoking last on a non empty sequence returns the element at the end of the sequence.

- **Length Axiom**
  - sequence(init.length) = 0

    The length of an empty sequence is zero.
- **Empty Axioms**
  - sequence(init.empty)= true

    An initial sequence is empty.
  - sequence(init.putop(a).empty)= false

    A sequence in which an element has been putted on it is not empty.

2. ***Rules***. We propose the following rules.

- **Init Rule**
  - sequence(h.init.h') = sequence(init.h')

    The init operation reinitializes the sequence, i.e. renders all past input history irrelevant.
- **Init Delete Rules**
  - sequence(init.deletehead.h) = sequence(init.h)

    A deletehead operation executed on an empty sequence has no effect.
  - sequence(init.deletelast.h) = sequence(init.h)

    A deletelast operation executed on an empty sequence has no effect.
- **Puthead Delete Rules**
  - sequence(init.h.puthead(a).deletehead.h^+) = sequence(init.h.h^+)

    A deletehead operation removes the first element putted in the sequence.
  - sequence(init.puthead(a).deletelast.h^+) = sequence(init.h^+)

    A deletelast operation removes the first element putted in the sequence if and only if there are no other elements than it.
- **Putlast Delete Rules**
  - sequence(init.h.putlast(a).deletelast.h^+) = sequence(init.h.h^+)

    A deletelast operation removes the last element putted in the sequence.
  - sequence(init.putlast(a).deletehead.h^+) = sequence(init.h^+)

    A deletehead operation removes the last element putted in the sequence if and only if there are no other elements than it.

- **Length Rule**
  - sequence(init.h.putop(a).length) =1+ sequence(init.h.length)

    Putop( ) increases the size of the sequence by 1.
- **Empty Rules**
  - sequence(init.h.putop(a).h'.empty) $\Rightarrow$ sequence(init.h.h'.empty)
  - sequence(init.h.empty) $\Rightarrow$ sequence(init.h.deletehead.empty)
  - sequence(init.h.empty) $\Rightarrow$ sequence(init.h.deletelast.empty)

    Removing any put operation or adding a delete operation to the input history of a sequence makes it more empty.
- **VX-Operation Rules**
  - sequence(init.h.head.h^+) = sequence(init.h.h^+)
  - sequence(init.h.last.h^+) = sequence(init.h.h^+)
  - sequence(init.h.length.h^+) = sequence(init.h.h^+)
  - sequence(init.h.empty.h^+) = sequence(init.h.h^+)

    VX operations leave no trace of their passage; once they are serviced and another follows them, they are forgotten: whether they occurred or did not occur has no impact on the future behavior of the sequence.

2. SPECIFICATION OF SET

We discuss how to represent the specification of a set, in the same way that we wrote the specification of sequence above. We represent in turn, the input space (from which we infer the set of input histories), then the output space, then the relation, which we denote by set.

1. *Input Space*. We let X be defined as:

X = {init, search, choose, smallest, largest, list, size, empty} ∪ {insert, remove} × itemtype.

We partition this set into O*X* = {init, insert, remove} and *VX* = {choose, smallest, largest, list, size, search, empty}. We let *H* be the set of sequences of elements of *X*.

2. *Output Space*. We let the output space be defined as:

   Y = {error} ∪ itemtype ∪ +∞ ∪ -∞ ∪ itemtype* ∪ emptyset ∪ integer ∪ Boolean.

1. ***Axioms***. We propose the following axioms:

- **Size Axiom**
  - set(init.size)= 0

    The size of an empty set is zero.

- **Empty Axioms**
  - set(init.empty)= true
  - set(init.insert(a).empty)= false

    An initial set is empty. A set in which an element has been inserted is not empty.
- **Search Axiom**
  - set(init.search(a)) = false

    The search of *a* in an empty set returns false.
- **Choose Axioms**
  - set(init.choose) = error

    One cannot choose an element from an empty set.
  - set(init.h.insert(a).choose)=a

    If *a* is an element of the set, then it is a possible choose; we will add a Commutativity rule later to make it possible to choose other elements than the most recently inserted element.
- **List Axiom**
  - set(init.list) = emptyset

    Requesting a list of elements from an empty set returns empty.
- **Smallest Axiom**
  - set(init.smallest) = $+\infty$.

    Plus infinity is the neutral element of operation smallest.
- **Largest Axiom**
  - set(init.largest) = $-\infty$.

    Minus infinity is the neutral element of operation largest.

2. ***Rules***. We propose the following rules.

- **Init Rule**
  - set(h.init.h') = set(init.h')

    Operation init makes the previous history irrelevant.
- **Init Remove Rule**
  - set(init.remove(a).h) = set(init.h)

    A remove operation has no effect on an empty set.
- **Insert Remove Rule**
  - set(init.h.insert(a).remove(a).h+) = set(init.h.h+)

    Operation remove cancels the effect of operation insert.
- **Size Rule**
  - **if** (set(init.h.search(a))) **then** set(init.h.insert(a).size) = set(init.h.size)

else set(init.h.insert(a).size) = 1 + set(init.h.size)

Operation insert(a) increases the size of the set only if element *a* is not in the set prior to insertion.

- **Search Rule**
  - set(init.h.insert(a).search(b)) = (a=b) set(init.h.search(b))

    If (a=b) then return true (since *b* is found in the set), else check prior to the insertion of *a*.
- **List Rule**
  - set(init.h.insert(a).list) = {a} ∪ set(init.h.list)

    If *a* has been inserted, it should be listed.
- **Commutativity Rule**
  - set(init.h.op1(a).op2(b).h')=set(init.h.op2(b).op1(a).h').

    where op1 and op2 are any of insert and remove operations and *a* and *b* are distinct. Interpretation: the order of operations of insert and remove of distinct elements is immaterial.
- **Insert Rule**
  - set(init.h.insert(a).insert(a).h') = set(init.h.insert(a).h')

    if *a* is already inserted in the set, it should not be inserted again.
- **Smallest Rule**
  - set(init.h.insert(a).smallest) = MIN(a , set(init.h.smallest))

    Inductive argument on the min.
- **Largest Rule**
  - set(init.h.insert(a).largest) = MAX(a , set(init.h.largest))

    Inductive argument on the max.
- **Empty Rules**
  - set(init.h.insert(a).h'.empty) $\Rightarrow$ set(init.h.h'.empty)
  - set(init.h.empty) $\Rightarrow$ set(init.h. remove(a).empty)

    Removing an insert or adding a remove operation to the input history of a set makes it more empty.
- **VX –Operation Rules**
  - set(init.h.size.h+) = set(init.h.h+)
  - set(init.h.empty.h+) = set(init.h.h+)
  - set(init.h.search.h+) = set(init.h.h+)
  - set(init.h.choose.h+) = set(init.h.h+)
  - set(init.h.list.h+) = set(init.h.h+)
  - set(init.h.min.h+) = set(init.h.h+)

- set(init.h.max.h+) = set(init.h.h+)
  VX operations leave no trace of their passage once they have been passed.

3. SPECIFICATION OF MULTISET

We discuss how to represent the specification of a multiset, in the same way that we wrote the specification of sequence and set above. We represent in turn, the input space (from which we infer the set of input histories), then the output space, then the relation, which we denote by multiset.

1. *Input Space*. We let X be defined as:

X = {init, removeany, eraseany ,search, multisearch, choose, smallest, largest, list, list(n), size, multisize, empty, least, most}∪{insert, insert(x, n), remove, remove(x, n), removeall}× itemtype. We partition this set into O*X* = {init, insert, insert(x, n), remove, remove(x, n), removeall, removeany, eraseany } and *VX* = { choose, least, most , smallest, largest, list, list(n), size, multisize, multisearch, empty, search }. We let *H* be the set of sequences of elements of *X*.

2. *Output Space*. We let the output space be defined as:

   Y = {error} ∪ itemtype ∪ +∞ ∪ -∞ ∪ itemtype* ∪ emptymultiset ∪ integer ∪ Boolean.

1. Axioms. We propose the following axioms:

- **Size Axiom**
  - multiset(init.size)= 0
    The size of an empty multiset is zero.
- **Multisize Axiom**
  - multiset(init.multisize)= 0
    The multisize of an empty multiset is zero.
- **Empty Axioms**
  - multiset(init.empty)= true
    An initial multiset is empty.
  - multiset(init.insertop(a).empty)= false
    Where insertop( ) is any insert or insert(x, n) operation. Interpretation: A multiset in which an element has been inserted is not empty.
- **Search Axiom**
  - multiset(init.search(a))= false
    The search of *a* in an empty multiset returns false.
- **Multisearch Axiom**
  - multiset(init.multisearch(a))= 0

Requesting the multiplicity of an element *a* in an empty multiset returns zero.

- **Choose Axioms**
  - multiset(init.choose) = error

    One cannot choose an element from an empty multiset.
  - multiset(init.h.insertop(a).choose)=a

    If *a* is an element of the multiset, then it is a possible choose; we will add a Commutativity rule later to make it possible to choose other elements than the most recently inserted element.
- **List Axiom**
  - multiset(init.list) = emptymultiset

    Requesting a list of elements from empty multiset returns empty.
- **List(n) Axiom**
  - multiset(init.list(n)) = emptymultiset

    Requesting a list of elements with a multiplicity greater than or equal to *n* from an empty multiset returns empty.
- **Smallest Axiom**
  - multiset(init.smallest)= + ∞

    Plus infinity is the neutral element of operation smallest.
- **Largest Axiom**
  - multiset(init.largest)= - ∞

    Minus infinity is the neutral element of operation largest.
- **Least Axioms**
  - multiset (init.least)= error

    Requesting the element with the minimal multiplicity from an empty multiset returns error.
  - multiset (init.insertop(a).insertop(_)*. least)= a , (_)*

    where insertop(_)*is any insert operation including zero(no operation). Interpretation: if more than one element has the same multiplicity that is minimal one then Invoking least returns all this elements.
- **Most Axioms**
  - multiset (init.most)= error

    Requesting the element with the maximum multiplicity from an empty multiset returns error.
  - multiset (init.insertop(a).insertop(_)*.most)= a , (_)*

    if more than one element has the same multiplicity that is maximum one then Invoking most returns all this elements.

2. ***Rules***. We propose the following rules.

- **Init Rule**
  - multiset (h.init.h') = multiset (init.h')

    Operation init makes the previous history irrelevant.
- **Init Remove Rules**
  - multiset (init.remove(a).h) = multiset (init.h)
  - multiset (init.remove(a, n).h) = multiset (init.h)
  - multiset (init.removeall(a).h) = multiset (init.h)
  - multiset (init.removeany.h) = multiset (init.h)
  - multiset (init.eraseany.h) = multiset (init.h)

    A remove operation has no effect on an empty multiset.
- **Insert Remove Rules**
  - multiset (init.h.insert(a).remove(a).h+) = multiset (init.h.h+)

    Operation remove cancels the effect of operation insert.
  - multiset (init.h.insert(a, n).remove(a).h+) = multiset (init.h.insert(a,n-1).h+)

    Operation remove reduce the multiplicity of an element *a* by 1.
- **Insert Remove(x, n) Rule**
  - **If** (multiset (init.h.multisearch(a)) <= n) **then**

    multiset (init.h.insertop(a)^{m}.remove(a, n).h+) = multiset (init.h.h+)

    else multiset (init.h.insertop(a)^{m}.remove(a, n).h+) =

    multiset (init.h.insertop(a)$^{m-n}$ .h+)

    Where insertop(a)^{m} is any insert operation of element *a* repeated *m* times. Interpretation: if the multiplicity of an element *a* is less than or equal *n* then the operation remove(a, n) deletes all instances of *a* else it reduce its multiplicity by *n*.
- **Insert Removeall Rule**
  - **If** (multiset (init.h.search(a)) **then**

    multiset (init.h.insertop(a)*.removeall(a).h+) = multiset (init.h.h+)

    else multiset (init.h.insertop(_)*.removeall(a).h+) =

    multiset(init.h.insertop(_)*.h+)

    Operation removeall deletes all instances of an element *a* if it is inserted in the multiset else it has no effect.
- **Insert Removeany Rules**
  - multiset (init.h.insert(a).removeany.h+) = multiset (init.h.h+)

    Operation removeany deletes an arbitrary element of multiset.
  - multiset (init.h.insert(a, n).removeany.h+) = multiset (init.insert(a, n-1).h.h+)

Operation removeany deletes an arbitrary element of multiset, reducing its multiplicity by 1.

- **Insert Eraseany Rule**
  - multiset (init.h.insertop(a).eraseany.h+) = multiset (init.h.h+)

    Operation eraseany removes all instances of an arbitrary element of multiset.
- **Size Rule**
  - **if** (multiset(init.h.search(a)) **then**

    multiset(init.h.insertop(a).size)=multiset(init.h.size)

    else multiset(init.h.insertop(a).size) = 1 + multiset(init.h.size).

    Any insert( ) operation increases the size of the multiset by 1 if and only if element *a* is not in the multiset prior to insertion.
- **Multisize Rules**
  - multiset (init.h.insert(a).multisize) = 1+ multiset (init.h.multisize)

    Operation insert( ) increases the size of the multiset by 1 if element *a* is in the multiset prior to insertion or not.
  - multiset (init.h.insert(a, n).multisize) = n+ multiset(init.h.multisize)

    Operation insert(x, n) increases the size of the multiset by n if element *a* is in the multiset prior to insertion or not.
- **Search Rule**
  - multiset(init.h.insertop(a).search(b)) = (a=b) multiset(init.h.search(b))

    If (a=b) then return true (since *b* is found in the multiset), else check prior to the insertion of *a*.
- **Multisearch Rules**
  - multiset(init.h.insert(a).insertop(_)*.multisearch(a)) = 1 + multiset(init.h.insertop(_)*.multisearch(a))

    Operation insert(a) increases the multiplicity of the element *a* by 1 if the element a is already inserted in the multiset.
  - multiset(init.h.insert(a, n).insertop(_)*.multisearch(a)) = n + multiset(init.h.insertop(_)*.multisearch(a))

    Operation insert(a, n) increases the multiplicity of the element *a* by n if the element a is already inserted in the multiset.
  - multiset(init.h. insertop(b).multisearch(a)) = multiset(init.h.multisearch(a))

    searching a non-existing element in the multiset will return nothing.
- **List Rules**
  - multiset (init.h.insert(a).list) = {a} ∪ multiset (init.h.list)

    If *a* has been inserted, it should be listed.

- multiset (init.h.insert(a, n).list) = {a : n} ∪ multiset (init.h.list)

  If *a* has been inserted *n* times, it should be listed *n* times.

- **List(n) Rule**
  - **if** (multiset(init.h.multisearch(a)) >= n) **then**

    multiset(init.h.insertop(a).list(n)) = {a} ∪ multiset (init.h.list(n))

    else multiset (init.h.insertop(a).list(n)) = multiset (init.h.list(n))

    If *a* has been inserted many times that are greater than or equal to *n* ,then it should be listed.

- **Commutativity Rule**
  - multiset(init.h.op1(a).op2(b).h')=multiset(init.h.op2(b).op1(a).h')

    Where op1 and op2 are any of insert and remove operations and *a* and *b* are distinct. Interpretation: the order of operations of insert and remove of distinct elements is immaterial.

- **Smallest Rule**
  - multiset(init.h.insertop(a).smallest) = MIN(a , multiset(init.h.smallest))

    Inductive argument on the min.

- **Largest Rule**
  - multiset(init.h.insertop(a).largest) = MAX(a , multiset(init.h.largest))

    Inductive argument on the max.

- **Least Rule**
  - **If** (multiset(init.h.multisearch(a)) < multiset(init.h.multisearch(b))) **then**

  multiset(init.h.insertop(a).insertop(b).least)= multiset (init.h.insertop(a).least)

  else

  If (multiset(init.h.multisearch(a)) = multiset(init.h.multisearch(b))) **then**

  multiset(init.h.insertop(a).insertop(b).least)= multiset (init.h.insertop(a) insertop(b).least)

  else multiset(init.h.insertop(a).insertop(b).least)= multiset(init.h.insertop(b).least)

- **Most Rule**
  - **If** (multiset(init.h.multisearch(a)) > multiset(init.h.multisearch(b)) **then**

  multiset(init.h.insertop(a).insertop(b).most) = multiset (init.h.insertop(a).most)

  else

  If (multiset(init.h.multisearch(a)) = multiset(init.h.multisearch(b)) **then**

  multiset(init.h.insertop(a).insertop(b).most)=

  multiset(init.h.insertop(a).insertop(b).most)

  else multiset(init.h.insertop(a).insertop(b).most)=multiset(init.h.insertop(b).most)

- **Empty Rules**
  - multiset (init.h.insertop(a).h'.empty) $\Rightarrow$ multiset (init.h.h'.empty)
  - multiset (init.h.empty) $\Rightarrow$ multiset (init.h.remove(a).empty)
  - multiset (init.h.empty) $\Rightarrow$ multiset (init.h. remove(a, n).empty)
  - multiset (init.h.empty) $\Rightarrow$ multiset (init.h. removeall(a).empty)
  - multiset (init.h.empty) $\Rightarrow$ multiset (init.h. removeany.empty)
  - multiset (init.h.empty) $\Rightarrow$ multiset (init.h. eraseany.empty)

  Removing any insert or adding any remove operations to the input history of a multiset makes it more empty.
- **VX– Operation Rules**
  - multiset (init.h.size.h+) = multiset (init.h.h+)
  - multiset (init.h.multisize.h+) = multiset (init.h.h+)
  - multiset (init.h.empty.h+) = multiset (init.h.h+)
  - multiset (init.h.search(a).h+) = multiset (init.h.h+)
  - multiset (init.h.multisearch(a).h+) = multiset (init.h.h+)
  - multiset (init.h.choose.h+) = multiset (init.h.h+)
  - multiset (init.h.list.h+) = multiset (init.h.h+)
  - multiset (init.h.list(n).h+) = multiset (init.h.h+)
  - multiset (init.h.least.h+) = multiset (init.h.h+)
  - multiset (init.h.most.h+) = multiset (init.h.h+)
  - multiset (init.h.smallest.h+) = multiset (init.h.h+)
  - multiset (init.h.largest.h+) = multiset (init.h.h+)

  VX operations leave no trace of their passage once they have been passed.

4. SPECIFICATION OF LIST

We discuss how to represent the specification of a list, in the same way that we wrote the specification of sequence, set and multiset above. We represent in turn, the input space (from which we infer the set of input histories), then the output space, then the relation, which we denote by list.

1. *Input Space*. We let X be defined as:

X = {init, deletefirst, deletelast, deleteat, search, multisearch, choose, first, last, smallest, largest, size, empty} ∪ { insertlast, insertfirst, insertat }× itemtype. We partition this set into O*X* = {init, insertlast, insertfirst, insertat, deletefirst, deletelast, deleteat} and *VX* = {choose,

first, last, smallest, largest, size, multisearch, empty, search}. We let *H* be the set of sequences of elements of *X*.

2. *Output Space*. We let the output space be defined as:

   Y = {error} ∪ itemtype ∪ +∞ ∪ -∞ ∪ integer ∪ Boolean.

1. ***Axioms***. We propose the following axioms:

   - **Size Axiom**
     - list(init.size)= 0

       The size of an empty list is zero.
   - **Empty Axioms**
     - list(init.empty)= true

       An initial list is empty.
     - list(init.insertlast(a).empty)= false

       A list in which an element has been inserted is not empty.
   - **Search Axiom**
     - list(init.search(a))= false

       The search of *a* in an empty list returns false.
   - **Multisearch Axiom**
     - list(init.multisearch(a))= 0

       Requesting the multiplicity of an element *a* in an empty list returns zero.
   - **Choose Axioms**
     - list(init.choose) = error

       One cannot choose an element from an empty list.
     - list(init.h.insertlast(a).choose)=a

       If *a* is an element of the list, then it is a possible choose; we will add a Commutativity rule later to make it possible to choose other elements than the most recently inserted element.
   - **First Axioms**
     - list(init.first)= error

       Invoking first on an empty list returns an error.
     - list(init.insertlast(a).insertlast(_)*.first)= a

       Invoking first on a non empty list returns the first element inserted into the list.
   - **Last Axioms**
     - list(init.last)= error

       Invoking last on an empty list returns an error.
     - list(init.insertlast(_)*.insertlast(a).last)= a

       Invoking last on a non empty list returns the last element inserted into the list.

- **Smallest Axiom**
  - list(init.smallest) = + ∞

    Plus infinity is the neutral element of operation smallest.
- **Largest Axiom**
  - list(init.largest) = - ∞

    Minus infinity is the neutral element of operation largest.

2. ***Rules***. We propose the following rules.

- **Init Rule**
  - list(h.init.h') = list(init.h')

    Operation init makes the previous history irrelevant.
- **Init Delete Rules**
  - list(init.deletefirst.h) = list(init.h)
  - list(init.deletelast.h) = list(init.h)
  - list(init.deleteat(n).h) = list(init.h)

    Any delete operation has no effect on an empty list.
- **Insertlast Delete Rules**
  - list(init.insertlast(a).insertlast(_)*.deletefirst.h+) = list(init.insertlast(_)*.h+)

    Operation deletefirst remove the first element in the list.
  - list(init.insertlast(_)*.insertlast(a).deletelast.h+) = list(init.insertlast(_)*.h+)

    Operation deletelast remove the last element in the list.
  - **If** (list(init.h.h'.size) >= n) then

    list(init.h.insertlast(a).deleteat(n).h') = list(init.h.h')

    else list(init.h.insertlast(a).deleteat(n).h') = list(init.h.insertlast(a).h')

    If the size of the list allows, operation deleteat(n) remove the element at the position *n* in the list, else it has no effect.
- **Search Rule**
  - list(init.h.insertlast(a).search(b))= (a=b) list(init.h.search(b))

    If (a=b) then return true (since *b* is found in the list), else check prior to the insertion of *a*.
- **Multisearch Rules**
  - list(init.h.insertlast(a).insertlast(_)*.multisearch(a)) = 1 +

    list(init.h.insertlast(_)*.multisearch(a))

    Operation insertlast( ) increases the multiplicity of the element *a* by 1 if the element *a* is already inserted in the list.
  - list(init.h.insertlast(b).multisearch(a))=list(init.h.multisearch(a))

- **Size Rule**
    - list(init.h.insertlast(a).size) = 1+ list(init.h.size)

      insertlast( ) increases the size of the list by 1.
- **Convert Rules**
    - list(init.h.insertfirst(a).h') = list(init.h.insertlast(a).h')
    - **If** (list(init.h.h'.size) < n) then

      list(init.h.insertat(a, n).h') = list(init.h.h')

      else list(init.h.insertat(a, n).h') = list(init.h.insertlast(a).h')
- **Smallest Rule**
    - list(init.h.insertlast(a).smallest) = MIN(a , list(init.h.smallest))

      Inductive argument on the min.
- **Largest Rule**
    - list(init.h.insertlast(a).largest) = MAX(a , list(init.h.largest))

      Inductive argument on the max.
- **Empty Rules**
    - list(init.h.insertlast(a).h'.empty) $\Rightarrow$ list(init.h.h'.empty)
    - list(init.h.empty) $\Rightarrow$ list(init.h. deletefirst.empty)
    - list(init.h.empty) $\Rightarrow$ list(init.h. deletelast.empty)
    - list(init.h.empty) $\Rightarrow$ list(init.h. deleteat(n).empty)

      Removing insert or adding any delete operations to the input history of a list makes it more empty.
- **VX–Operation Rules**
    - list(init.h.size.h+) = list(init.h.h+)
    - list(init.h.empty.h+) = list(init.h.h+)
    - list(init.h.search(a).h+) = list(init.h.h+)
    - list(init.h.multisearch(a).h+) = list(init.h.h+)
    - list(init.h.choose.h+) = list(init.h.h+)
    - list(init.h.first.h+) = list(init.h.h+)
    - list(init.h.last.h+) = list(init.h.h+)
    - list(init.h.smallest.h+) = list(init.h.h+)
    - list(init.h.largest.h+) = list(init.h.h+)

      VX operations leave no trace of their passage once they have been passed.

APPENDIX -C

THE VALIDATION DATA FOR ABSTRACT DATA TYPES

1. THE VALIDATION DATA FOR QUEUE DATA TYPE

1. queue(init.enqueue(a).enqueue(b).front.dequeue.size) = **1**

={ by virtue of VX-op rule }

queue(init.enqueue(a).enqueue(b).dequeue.size)

={ by virtue of enqueue dequeue rule }

queue(init.enqueue(b).size)

={ by virtue of the size rule, with h=< > }

1 + queue(init.size)

={ by virtue of the size axiom}

1 + 0

= {arithmetic}

1. QED

2. queue(init.enqueue(a).dequeue.enqueue(b).enqueue(c).empty.rear) = **c**

={ by virtue of VX-op rule }

queue(init.enqueue(a).dequeue.enqueue(b).enqueue(c).rear)

={ by virtue of enqueue dequeue rule, with enqueue(_)*= < > }

queue(init.enqueue(b).enqueue(c).rear)

={ by virtue of the second rear axiom}

c. QED

3. queue(init.enqueue(a).enqueue(b).front.init.dequeue.enqueue(a).dequeue.size) = 0

={ by virtue of init rule }

queue(init.dequeue.enqueue(a).dequeue.size)

={ by virtue of init dequeue rule }

queue(init.enqueue(a).dequeue.size)

={ by virtue of enqueue dequeue rule, with enqueue(_)*= < > }

queue(init.size)

={ by virtue of size axiom }

0. QED

4. queue(init.enqueue(a).enqueue(b).front.init.dequeue.enqueue(a).dequeue.empty.empty)= **true**

={ by virtue of init rule }

queue(init.dequeue.enqueue(a).dequeue.empty.empty)

={ by virtue of init dequeue rule }

queue(init.enqueue(a).dequeue.empty.empty)

={ by virtue of VX-op rule }

queue(init.enqueue(a).dequeue.empty)

={ by virtue of enqueue dequeue rule, with enqueue(_)*= < > }

queue(init.empty)

={ by virtue of the first empty axiom }

true. QED

5. queue(init.enqueue(a).enqueue(b).front.init.dequeue.dequeue.size.enqueue(c).size.front) = **c**

={ by virtue of init rule }

queue(init.dequeue.dequeue.size.enqueue(c).size.front)

={ by virtue of VX-op rule }

queue(init.dequeue.dequeue.enqueue(c).front)

={ by virtue of init dequeue rule, applied twice }

queue(init.enqueue(c).front)

={ by virtue of the second front axiom, with enqueue(_)*= < > }

c. QED

6. queue(init.enqueue(a).enqueue(b).front.init.dequeue.dequeue.init.size.front) = **error**

={ by virtue of init rule, applied twice}

queue(init.size.front)

={ by virtue of VX-op rules }

queue(init.front)

={ by virtue of the first front axiom}

error. QED

2. THE VALIDATION DATA FOR SEQUENCE DATA TYPE

1. sequence(init.puthead(a).puthead(b).putlast(c).deletelast.puthead(d).empty) = **false**

={ by virtue of the first putlast delete rule, with h=<puthead(a).puthead(b) }

sequence(init.puthead(a).puthead(b).puthead(d).empty)=

{ by virtue of the first empty rule , with h=<puthead(a) > ,h’=< puthead(d) >}

sequence(init.puthead(a).puthead(d).empty)

={ by virtue of the first empty rule , with h=<puthead(a)> ,h’=<>}

sequence(init.puthead(a).empty)

={ by virtue of the second empty axiom}

false. QED

2. sequence(init.puthead(a).puthead(b).putlast(c).init.puthead(d).length) = **1**

={ by virtue of the init rule }

sequence(init.puthead(d).length)

={ by virtue of the length rule , with h=<> }

1+ sequence(init.length)

={ by virtue of the length axiom}

1 + 0

= {arithmetic}

1. QED

3. sequence(init.puthead(a).puthead(b).init.putlast(c).puthead(d).head) = **d**

={ by virtue of the init rule }

sequence(init.putlast(c).puthead(d).head)

={ by virtue of the second head axiom}

d. QED

4. sequence(init.puthead(a).puthead(b).init.putlast(c).puthead(d).Last) = **c**

={ by virtue of the init rule }

sequence(init.putlast(c).puthead(d).Last)

={ by virtue of the third last axiom }

c. QED

5. sequence(init.puthead(a).puthead(b).init.putlast(c).puthead(d).deletehead.last) = **c**

={ by virtue of the init rule }

sequence(init.putlast(c).puthead(d).deletehead.last)

={ by virtue of the first puthead delete rule, with h = <putlast(c) > }

sequence(init.putlast(c).last)

={ by virtue of the second last axiom }

c. QED

6. sequence(init.puthead(a).puthead(b).puthead(c).deletehead.deletehead.deletehead.
puthead(d).putlast(e).deletelast.head) = **d**

={ by virtue of the first puthead delete rule ,applied three times }

sequence(init.puthead(d).putlast(e).deletelast.head)

={ by virtue of the first putlast delete rule, with h = < puthead(d) >}

sequence(init.puthead(d).head)

={ by virtue of the second head axiom}

d. QED

7. sequence(init.puthead(a).head.puthead(b).length.puthead(c).last.deletehead.deletehead.
empty.deletehead.length) = **0**

={ by virtue of VX-op rules }

sequence(init.puthead(a).puthead(b).puthead(c).deletehead.deletehead.deletehead. length)

={ by virtue of the first puthead delete rule ,applied three times }

sequence(init.length)

={ by virtue of the length axiom}

0. QED

8. sequence(init.puthead(a).puthead(b).puthead(c).deletehead.deletehead.deletehead.
deletelast.empty) = **true**

={ by virtue of the first puthead delete rule ,applied three times }

sequence(init.deletelast.empty)

={ by virtue of the init delete rule}

sequence(init.empty)

={ by virtue of the first empty axiom}

true. QED

3. THE VALIDATION DATA FOR SET DATA TYPE

1. set(init.insert(a).insert(b).remove(a).insert(c).init.empty) = **true**

= {by virtue of init rule}

set(init.empty)

= {by virtue of first empty axiom}

true. QED

2. set(init.insert(a).insert(b).remove(b).remove(a).insert(a).empty) = **false**

= {by virtue of the insert remove rule, with h = < insert(a) >}

set(init.insert(a).remove(a).insert(a).empty)

= {by virtue of the insert remove rule, with h = < >}

set(init.insert(a).empty)

= {by virtue of second empty axiom}

false. QED

set(init.insert(a).insert(b).remove(a).search(a).remove(b).choose.size) = **0**

= {by virtue of VX-op rule}

set(init.insert(a).insert(b).remove(a).remove(b).size)

= {by virtue of Commutativity rule, with h = < >}

set(init.insert(b).insert(a).remove(a).remove(b).size)

= {by virtue of insert remove rule, with h = < insert(b) >}

set(init.insert(b).remove(b).size)

= {by virtue of the insert remove rule, with h = < >}

set(init.size)

= {by virtue of size axiom}

0. QED

3. set(insert(a).list.insert(b).init.insert(a).insert(b).insert(c).remove(a).empty.insert(a).size) = **3**

= {by virtue of init rule}

set(init.insert(a).insert(b).insert(c).remove(a).empty.insert(a).size)

= {by virtue of VX-op rule}

set(init.insert(a).insert(b).insert(c).remove(a).insert(a).size)

= {by virtue of Commutativity rule, with h = < >}

set(init.insert(b).insert(a).insert(c).remove(a).insert(a).size)

= {by virtue of Commutativity rule, with h = < insert(b)>}

set(init.insert(b).insert(c).insert(a).remove(a).insert(a).size)

= {by virtue of insert remove rule, with h = < insert(b).insert(c)>}

set(init.insert(b).insert(c).insert(a).size)

= {by virtue of size rule, with h=< insert(b).insert(c) >}

1 + set(init.insert(b).insert(c).size)

= {by virtue of size rule, with h=< insert(b) >}

1 + 1 + set(init.insert(b).size)

= {by virtue of size rule, with h=<>}

1 + 1 + 1 + set(init.size)

= {by virtue of size axiom}

1 + 1 + 1 + 0

= {arithmetic}

3. QED

4. set(init.insert(a).insert(b).remove(a).insert(c).remove(b).insert(a).search(a)) = **true**

= {by virtue of Commutativity rule, with h = < >}

set(init.insert(b).insert(a).remove(a).insert(c).remove(b).insert(a).search(a))

= {by virtue of insert remove rule, with h=< insert(b) >}

set(init.insert(b).insert(c).remove(b).insert(a).search(a))

= {by virtue of Commutativity rule, with h = < >}

set(init.insert(c).insert(b).remove(b).insert(a).search(a))

= {by virtue of insert remove rule, with h=< insert(c) >}

set(init.insert(c).insert(a).search(a))

= {by virtue of the search rule, with h = < insert(c) >}

true. QED

5. set(insert(a).insert(b).remove(b).choose.init.insert(a).insert(b).empty.remove(b). choose) = **a**

= {by virtue of init rule}

set(init.insert(a).insert(b).empty.remove(b).choose)

= {by virtue of VX-op rule}

set(init.insert(a).insert(b).remove(b).choose)

= {by virtue of the insert remove rule, with h = <insert(a) >}

set(init.insert(a).choose)

= {by virtue of second choose axiom, with h = <>}

a. QED

6. set(init.insert(a).insert(b).insert(c).smallest.insert(d).remove(b).remove(d).list) = **{a, c}**

= {by virtue of VX-op rule}

set(init.insert(a).insert(b).insert(c).insert(d).remove(b).remove(d).list)

= {by virtue of Commutativity rule, with h = < insert(a) >}

set(init.insert(a).insert(c).insert(b).insert(d).remove(b).remove(d).list)

= {by virtue of Commutativity rule, with h = < insert(a).insert(c) >}

set(init.insert(a).insert(c).insert(d).insert(b).remove(b).remove(d).list)

= {by virtue of insert remove rule, with h=< insert(a).insert(c).insert(d)>}

set(init.insert(a).insert(c).insert(d).remove(d).list)

= {by virtue of the insert remove rule with h=< insert(a).insert(c) >}

set(init.insert(a).insert(c).list)

= {by virtue of Commutativity rule, with h = < >}

set(init.insert(c).insert(a).list)

= {by virtue of list rule, with h=< insert(c) >}

{a} $\cup$ set(init.insert(c).list)

= {by virtue of list rule, with h=< >}

{a} $\cup$ {c} $\cup$ set(init.list)

= {by virtue of the list axiom}

{a} $\cup$ {c} $\cup$ emptyset

= {set theory}

{a ,c}. QED

7. set(init.insert(a).insert(a).remove(a).insert(c).remove(c).insert(a).list) = **{a}**

= {by virtue of insert rule, with h=< >}

set(init.insert(a).remove(a).insert(c).remove(c).insert(a).list)

= {by virtue of insert remove rule, with h = < >}

set(init.insert(c).remove(c).insert(a).list)

= {by virtue of insert remove rule, with h = < >}

set(init.insert(a).list)

= {by virtue of list rule, with h = < >}

{a} ∪ set(init.list)

= {by virtue of list axiom}

{a} ∪ emptyset

{set theory}

{a}. QED

8. set(init.insert(3).insert(2).choose.insert(1).remove(2).insert(4).remove(1).smallest) = **3**

= {by virtue of VX-op rule}

set(init.insert(3).insert(2).insert(1).remove(2).insert(4).remove(1).smallest)

= {by virtue of Commutativity rule, with h=< insert(3) >}

set(init.insert(3).insert(1).insert(2).remove(2).insert(4).remove(1).smallest)

= {by virtue of insert remove rule, with h=< insert(3).insert(1) >}

set(init.insert(3).insert(1).insert(4).remove(1).smallest)

= {by virtue of Commutativity rule, with h=< insert(3) >}

set(init.insert(3).insert(4).insert(1).remove(1).smallest)

= {by virtue of insert remove rule, with h=< insert(3).insert(4) >}

set(init.insert(3).insert(4).smallest)

= {by virtue of smallest rule, with h=< insert(3)>}

Min (4 , set(init.insert(3).smallest))

= {by virtue of smallest rule, with h=< >}

Min (4 , 3, set(init.smallest))

= {by virtue of smallest axiom}

Min (4 , 3, +∞))

= {arithmetic}

3. QED

9. set(insert(5).insert(7).remove(5).init.insert(4).insert(5).search(7).insert(3).remove(4).largest)=**5**

= {by virtue of init rule}

set(init.insert(4).insert(5).search(7).insert(3).remove(4).largest)

= {by virtue of VX-op rule}

set(init.insert(4).insert(5).insert(3).remove(4).largest)

= {by virtue of Commutativity rule, with h=< >}

set(init.insert(5).insert(4).insert(3).remove(4).largest)

= {by virtue of Commutativity rule, with h=< insert(5) >}

set(init.insert(5).insert(3).insert(4).remove(4).largest)

= {by virtue of insert remove rule, with h=<insert(5).insert(3)>}

set(init.insert(5).insert(3).largest)

= {by virtue of largest rule, with h=< insert(5) >}

Max (3, set(init.insert(5).largest))

= {by virtue of largest rule, with h=< >}

Max (3, 5, set(init.largest))

= {by virtue of largest axiom}

Max (3, 5, -∞)

5. QED

4. THE VALIDATION DATA FOR MULTISET DATA TYPE

1. multiset(init.insert(a).insert(b).insert(c,3).insert(b).remove(a).removeall(b).remove(c,2). empty) = **false**

= {by virtue of Commutativity rule, with h= < >}

multiset(init.insert(b).insert(a).insert(c,3).insert(b).remove(a).removeall(b).remove(c,2). empty)

= {by virtue of Commutativity rule, with h= < insert(b) >}

multiset(init.insert(b).insert(c,3).insert(a).insert(b).remove(a).removeall(b).remove(c,2). empty)

= {by virtue of Commutativity rule, with h= < insert(b).insert(c,3) >}

multiset(init.insert(b).insert(c,3).insert(b).insert(a).remove(a).removeall(b).remove(c,2). empty)

= {by virtue of the first insert remove rule, with h=< insert(b).insert(c,3). insert(b)>}

multiset(init.insert(b).insert(c,3).insert(b).removeall(b).remove(c,2).empty)

= {by virtue of the insert removeall rule}

multiset(init.insert(c,3).remove(c,2).empty)

= {by virtue of the insert remove(x,n) rule}

multiset(init.insert(c).empty)

= {by virtue of the second empty axiom}

false. QED

2. multiset(init.insert(a,4).insert(b).eraseany.init.insert(a).insert(a,2).insert(b).removeall(a). removeany.empty) = **true**

= {by virtue of init rule}

multiset(init.insert(a).insert(a,2).insert(b).removeall(a).removeany.empty)

= {by virtue of Commutativity rule, with h= < insert(a)>}

multiset(init.insert(a).insert(b).insert(a,2).removeall(a).removeany.empty)

= {by virtue of Commutativity rule, with h= < >}

multiset(init.insert(b).insert(a).insert(a,2).removeall(a).removeany.empty)

= {by virtue of the insert removeall rule}

multiset(init.insert(b).removeany.empty)

= {by virtue of the first insert removeany rule, with h=<>}

multiset(init.empty)

= {by virtue of the first empty axiom}

true. QED

3. multiset(init.insert(a,5).insert(b).insert(a).insert(c,3).remove(a).search(a).remove(a).size) = **3**

= {by virtue of VX-op rules}

multiset(init.insert(a,5).insert(b).insert(a).insert(c,3).remove(a).remove(a).size)

= {by virtue of Commutativity rule, with h= < insert(a,5).insert(b) >}

multiset(init.insert(a,5).insert(b).insert(c,3).insert(a).remove(a).remove(a).size)

= {by virtue of the first insert remove rule, with h=< insert(a,5).insert(b). insert(c,3)>}

multiset(init.insert(a,5).insert(b).insert(c,3).remove(a).size)

= {by virtue of Commutativity rule, with h= < >}

multiset(init.insert(b).insert(a,5).insert(c,3).remove(a).size)

= {by virtue of Commutativity rule, with h= < insert(b) >}

multiset(init.insert(b).insert(c,3).insert(a,5).remove(a).size)

= {by virtue of the second insert remove rule, with h=< insert(b).insert(c,3)>}

multiset(init.insert(b).insert(c,3).insert(a,4).size)

= {by virtue of the size rule, with h=< insert(b).insert(c,3) >}

1 + multiset(init.insert(b).insert(c,3).size)

= {by virtue of the size rule, with h=< insert(b) >}

1 + 1 + multiset(init.insert(b).size)

= {by virtue of the size rule, with h=< >}

1 + 1 + 1 + multiset(init.size)

= {by virtue of size axiom}

1 + 1 + 1 + 0

= {arithmetic}

3. QED

4. multiset(init.insert(a).insert(b,4).insert(c,3).insert(a).remove(b).remove(c,2).choose. multisearch(a).size.insert(c).multisize) = **7**

= {by virtue of VX-op rules}

multiset(init.insert(a).insert(b,4).insert(c,3).insert(a).remove(b).remove(c,2).insert(c). multisize)

= {by virtue of Commutativity rule, with h= < insert(a).insert(b,4) >}

multiset(init.insert(a).insert(b,4).insert(a).insert(c,3).remove(b).remove(c,2).insert(c). multisize)

= {by virtue of Commutativity rule, with h= < insert(a).insert(b,4).insert(a) >}

multiset(init.insert(a).insert(b,4).insert(a).remove(b).insert(c,3).remove(c,2).insert(c). multisize)

= {by virtue of the insert remove (x,n) rule}

multiset(init.insert(a).insert(b,4).insert(a).remove(b).insert(c).insert(c).multisize)

= {by virtue of Commutativity rule, with h= < insert(a) >}

multiset(init.insert(a).insert(a).insert(b,4).remove(b).insert(c).insert(c).multisize)

= {by virtue of the second insert remove rule, with h =< insert(a).insert(a) >}

multiset(init.insert(a).insert(a).insert(b,3).insert(c).insert(c).multisize)

= {by virtue of the first multisize rule, with h=< insert(a).insert(a). insert(b,3).insert(c)>}

1 + multiset(init.insert(a).insert(a).insert(b,3).insert(c).multisize)

= {by virtue of the first multisize rule, with h=< insert(a).insert(a).insert(b,3) >}

1 + 1 + multiset(init.insert(a).insert(a).insert(b,3).multisize)

= {by virtue of the second multisize rule, with h=< insert(a).insert(a) >}

1 + 1 + 3 + multiset(init.insert(a).insert(a).multisize)

= {by virtue of the first multisize rule, with h=< insert(a) >}

1 + 1 + 3 + 1+ multiset(init.insert(a).multisize)

= {by virtue of the first multisize rule, with h=<>}

1 + 1 + 3 + 1 + 1 + multiset(init.multisize)

= {by virtue of multisize axiom}

1 + 1 + 3 + 1 + 1 + 0

= {arithmetic}

7. QED

5. multiset(init.insert(a,3).insert(b).most.insert(c).removeall(a).insert(b,2).least.insert(c). search(a)) = **false**

= {by virtue of VX-op rules}

multiset(init.insert(a,3).insert(b).insert(c).removeall(a).insert(b,2).insert(c).search(a))

= {by virtue of Commutativity rule, with h= < >}

multiset(init.insert(b).insert(a,3).insert(c).removeall(a).insert(b,2).insert(c).search(a))

= {by virtue of Commutativity rule, with h= < insert(b) >}

multiset(init.insert(b).insert(c).insert(a,3).removeall(a).insert(b,2).insert(c).search(a))

= {by virtue of the insert removeall rule}

multiset(init.insert(b).insert(c).insert(b,2).insert(c).search(a))

= {by virtue of the search rule}

multiset(init.insert(b).insert(c).insert(b,2).search(a))

= {by virtue of the search rule}

multiset(init.insert(b).insert(c).search(a))

= {by virtue of the search rule}

multiset(init.insert(b).search(a))

= {by virtue of the search rule}

multiset(init.search(a))

= {by virtue of the search axiom}

false. QED

6. multiset(init.insert(a).insert(b,3).insert(c).removeany.search(a).insert(c,5).list.insert(a,6).remove(b).multisearch(a)) = **7**

= {by virtue of VX-op rules}

multiset(init.insert(a).insert(b,3).insert(c).removeany.insert(c,5).insert(a,6).remove(b).multisearch(a))

= {by virtue of the first insert removeany rule, with h = < insert(a).insert(b,3) >}

multiset(init.insert(a).insert(b,3).insert(c,5).insert(a,6).remove(b).multisearch(a))

= {by virtue of Commutativity rule, with h= < insert(a) >}

multiset(init.insert(a).insert(c,5).insert(b,3).insert(a,6).remove(b).multisearch(a))

= {by virtue of Commutativity rule, with h= < insert(a).insert(c,5) >}

multiset(init.insert(a).insert(c,5).insert(a,6).insert(b,3).remove(b).multisearch(a))

= {by virtue of the first insert remove rule, with h = < insert(a).insert(c,5). insert(a,6) >}

multiset(init.insert(a).insert(c,5).insert(a,6).insert(b,2).multisearch(a))

= {by virtue of the second multisearch rule, with h = < insert(a).insert(c,5)>}

6 + multiset(init.insert(a).insert(c,5).insert(b,2).multisearch(a))

= {by virtue of the first multisearch rule, with h = < >}

6 + 1 + multiset(init.insert(c,5).insert(b,2).multisearch(a))

= {by virtue of the third multisearch rule, with h = < insert(c,5) > }

6 + 1 + multiset(init.insert(c,5).multisearch(a))

= {by virtue of the third multisearch rule, with h = < > }

6 + 1 + multiset(init.multisearch(a))

= {by virtue of the multisearch axiom}

6 + 1 + 0

{arithmetic}

7. QED

7. multiset(init.insert(a,3).insert(b,2).insert(c).eraseany.list(2).insert(a).choose) = **b**

= {by virtue of VX-op rules}

multiset(init.insert(a,3).insert(b,2).insert(c).eraseany.insert(a).choose)

= {by virtue of the insert eraseany rule, with h = <insert(a,3).insert(b,2)>}

multiset(init.insert(a,3).insert(b,2).insert(a).choose)

= {by virtue of the Commutativity rule, with h = <insert(a,3)>}

multiset(init.insert(a,3).insert(a).insert(b,2).choose)

= {by virtue of the second choose axiom, with h = <insert(a,3).insert(a)>}

b. QED

multiset(init.insert(a).insert(b,5).remove(a).removeall(b).init.insert(a,2).insert(b).
insert(c,3).eraseany.list) = {b , c, c, c }

= {by virtue of init rule}

multiset(init.insert(a,2).insert(b).insert(c,3).eraseany.list)

= {by virtue of the Commutativity rule, with h = < >}

multiset(init.insert(b).insert(a,2).insert(c,3).eraseany.list)

= {by virtue of the Commutativity rule, with h = < insert(b) >}

multiset(init.insert(b).insert(c,3).insert(a,2).eraseany.list)

= {by virtue of the insert eraseany rule, with h = <insert(b).insert(c,3)>}

multiset(init.insert(b).insert(c,3).list)

= {by virtue of the Commutativity rule, with h = < >}

multiset(init.insert(c,3).insert(b).list)

= {by virtue of the first list rule, with h=< insert(c,3) >}

{b} ∪ multiset(init.insert(c,3).list)

= {by virtue of the second list rule, with h=< >}

{b} ∪ {c, c, c } ∪ multiset(init.list)

= {by virtue of the list axiom}

{b} ∪ {c, c, c } ∪ emptymultiset

{set theory}

{b , c, c, c }. QED

8. multiset(init.insert(a,3).insert(b).insert(b).remove(a).insert(c,4).multisearch(a).
remove(c,2). insert(a).remove(b).list(3)) = **{a}**

= {by virtue of VX-op rules}

multiset(init.insert(a,3).insert(b).insert(b).remove(a).insert(c,4).remove(c,2).insert(a).
remove(b).list(3))

= {by virtue of the insert remove(x,n) rule, with h= <insert(a,3).insert(b).insert(b).
remove(a)>}

multiset(init.insert(a,3).insert(b).insert(b).remove(a).insert(c,2).insert(a).remove(b).list(3))

= {by virtue of Commutativity rule, with h = < insert(a,3).insert(b) >}

multiset(init.insert(a,3).insert(b).remove(a).insert(b).insert(c,2).insert(a).remove(b).list(3))

= {by virtue of Commutativity rule, with h = < insert(a,3) >}

multiset(init.insert(a,3).remove(a).insert(b).insert(b).insert(c,2).insert(a).remove(b).list(3))

= {by virtue of the second insert remove rule , with h= < >}

multiset(init.insert(a,2).insert(b).insert(b).insert(c,2).insert(a).remove(b).list(3))

= {by virtue of Commutativity rule, with h = < insert(a,2).insert(b) >}

multiset(init.insert(a,2).insert(b).insert(c,2).insert(b).insert(a).remove(b).list(3))

= {by virtue of Commutativity rule, with h = < insert(a,2).insert(b).insert(c,2) >}

multiset(init.insert(a,2).insert(b).insert(c,2).insert(a).insert(b).remove(b).list(3))

= {by virtue of the first insert remove rule, with h=< insert(a,2).insert(b).insert(c,2).insert(a)>}

multiset(init.insert(a,2).insert(b).insert(c,2).insert(a).list(3))

= {by virtue of the list(n) rule}

{a} ∪ multiset(init.insert(b).insert(c,2).list(3))

= {by virtue of the list(n) rule, with h=< insert(b) >}

{a} ∪ multiset(init.insert(b).list(3))

= {by virtue of the list(n) rule, with h=< >}

{a} ∪ multiset(init.list(3))

= {by virtue of the list(n) axiom}

{a} ∪ emptymultiset

= {set theory}

{a}. QED

9. multiset(init.insert(a).insert(b,3).remove(a).insert(c,2).insert(a).remove(b,2).choose.
insert(b).least) = **a**

= {by virtue of VX-op rules}

multiset(init.insert(a).insert(b,3).remove(a).insert(c,2).insert(a).remove(b,2).insert(b). least)

= {by virtue of Commutativity rule, with h = < insert(a) >}

multiset(init.insert(a).remove(a).insert(b,3).insert(c,2).insert(a).remove(b,2).insert(b). least)

= {by virtue of the first insert remove rule , with h= <>}

multiset(init.insert(b,3).insert(c,2).insert(a).remove(b,2).insert(b).least)

= {by virtue of Commutativity rule, with h = < >}

multiset(init.insert(c,2).insert(b,3).insert(a).remove(b,2).insert(b).least)

= {by virtue of Commutativity rule, with h = < insert(c,2) >}

multiset(init.insert(c,2).insert(a).insert(b,3).remove(b,2).insert(b).least)

= {by virtue of the insert remove(x,n) rule}

multiset(init.insert(c,2).insert(a).insert(b).insert(b).least)

= {by virtue of the least rule}

multiset(init.insert(c,2).insert(a).least)

= {by virtue of the least rule}

multiset(init.insert(a).least)

= {by virtue of the second least axiom}

a. QED

10. multiset(init.insert(a,5).insert(b).removeall(a).insert(c,2).insert(a).insert(b,3).search(a). insert(a,2).eraseany.most) = **a, b**

= {by virtue of VX-op rules}

multiset(init.insert(a,5).insert(b).removeall(a).insert(c,2).insert(a).insert(b,3).insert(a,2). eraseany.most)

= {by virtue of Commutativity rule, with h = < >}

multiset(init.insert(b).insert(a,5).removeall(a).insert(c,2).insert(a).insert(b,3).insert(a,2). eraseany.most)

= {by virtue of the insert removeall rule, with h= < insert(b)>}

multiset(init.insert(b).insert(c,2).insert(a).insert(b,3).insert(a,2).eraseany.most)

= {by virtue of Commutativity rule, with h = < >}

multiset(init.insert(c,2).insert(b).insert(a).insert(b,3).insert(a,2).eraseany.most)

= {by virtue of Commutativity rule, with h = < insert(c,2) >}

multiset(init.insert(c,2).insert(a).insert(b).insert(b,3).insert(a,2).eraseany.most)

= {by virtue of Commutativity rule, with h = < insert(c,2).insert(a) >}

multiset(init.insert(c,2).insert(a).insert(b,3).insert(b).insert(a,2).eraseany.most)

= {by virtue of Commutativity rule, with h = < insert(c,2).insert(a).insert(b,3) >}

multiset(init.insert(c,2).insert(a).insert(b,3).insert(a,2).insert(b).eraseany.most)

= {by virtue of the insert eraseany rule, with=< insert(c,2).insert(a).insert(b,3).insert(a,2)>}

multiset(init.insert(c,2).insert(a).insert(b,3).insert(a,2).most)

= {by virtue of the most rule}

multiset(init.insert(c,2).insert(a).insert(b,3).insert(a,2).most)

= {by virtue of the most rule}

multiset(init.insert(a).insert(b,3).insert(a,2).most)

= {by virtue of the second most axiom}

a, b. QED

11. multiset(init.insert(3).insert(2,3).list.remove(2).insert(1,5).multisearch(2).removeall(1).size.insert(4).smallest) = **2**

= {by virtue of VX-op rules}

multiset(init.insert(3).insert(2,3).remove(2).insert(1,5).removeall(1).insert(4).smallest)

= {by virtue of the second insert remove rule, with h= < insert(3) >}

multiset(init.insert(3).insert(2,2).insert(1,5).removeall(1).insert(4).smallest)

= {by virtue of the insert removeall rule, with h= < insert(3).insert(2,2) >}

multiset(init.insert(3).insert(2,2).insert(4).smallest)

= {by virtue of the smallest rule, with h = <insert(3).insert(2,2)>}

Min (4 , multiset(init.insert(3).insert(2,2).smallest))

= {by virtue of the smallest rule, with h = <insert(3) >}

Min (4 , 2, multiset(init.insert(3).smallest))

= {by virtue of the smallest rule, with h = < >}

Min (4 , 2 , 3 , multiset(init.smallest))

= {by virtue of the smallest axiom}

Min (4 , 2 , 3 , $+\infty$)

{arithmetic}

2. QED

12. multiset(insert(1,3).insert(4).insert(6).init.eraseany.insert(3).insert(5,3).list(2).insert(1).remove(5,2).choose.insert(2).remove(5).largest) = **3**

= {by virtue of init rule}

multiset(init.eraseany.insert(3).insert(5,3).list(2).insert(1).remove(5,2).choose.insert(2).remove(5).largest)

= {by virtue of the fifth init remove rule}

multiset(init.insert(3).insert(5,3).list(2).insert(1).remove(5,2).choose.insert(2).remove(5).largest)

= {by virtue of VX-op rules}

multiset(init.insert(3).insert(5,3).insert(1).remove(5,2).insert(2).remove(5).largest)

= {by virtue of Commutativity rule, with h = < insert(3)>}

multiset(init.insert(3).insert(1).insert(5,3).remove(5,2).insert(2).remove(5).largest)

= {by virtue of the insert remove(x,n) rule, with h = < insert(3).insert(1) >}

multiset(init.insert(3).insert(1).insert(5).insert(2).remove(5).largest)

= {by virtue of Commutativity rule, with h = < insert(3).insert(1)>}

multiset(init.insert(3).insert(1).insert(2).insert(5).remove(5).largest)

= {by virtue of the first insert remove rule, with h=< insert(3).insert(1).insert(2)>}

multiset(init.insert(3).insert(1).insert(2).largest)

= {by virtue of the largest rule, with h = < insert(3).insert(1) >}

Max (2 , multiset(init.insert(3).insert(1).largest))

= {by virtue of the largest rule, with h = < insert(3) >}

Max (2 , 1 , multiset(init.insert(3).largest))

= {by virtue of the largest rule, with h = < >}

Max (2 , 1 , 3, multiset(init.largest))

= {by virtue of the largest axiom}

Max (2 , 1 , 3 , $-\infty$)

{arithmetic}

3. QED

5. THE VALIDATION DATA FOR LIST DATA TYPE

1. list(init.insertfirst(a).insertfirst(b).deletelast.size) = **1**

= {by virtue of the first convert rule}

list(init.insertlast(b).insertfirst(a).deletelast.size)

= {by virtue of the first convert rule}

list(init.insertlast(b).insertlast(a).deletelast.size)

= {by virtue of the second insertlast delete rule}

list(init.insertlast(b).size)

= {by virtue of the size rule, with h=< >}

1 + list(init.size)

= {by virtue of size axiom}

1 + 0

= {arithmetic}

1. QED

2. list(init.insertfirst(b).insertfirst(a).deletelast.deleteat(1).size) = **0**

= {by virtue of the first convert rule}

list(init.insertlast(a).insertfirst(b).deletelast.deleteat(1).size)

= {by virtue of the first convert rule}

list(init.insertlast(a).insertlast(b).deletelast.deleteat(1).size)

= {by virtue of the second insertlast delete rule}

list(init.insertlast(a).deleteat(1).size)

= {by virtue of the third insertlast delete rule, with h= <>}

list(init.size)

= {by virtue of size axiom}

0. QED

3. list(init.insertfirst(a).insertfirst(b).deletelast.last) = **b**

= {by virtue of the first convert rule}

list(init.insertlast(b).insertfirst(a).deletelast.last)

= {by virtue of the first convert rule}

list(init.insertlast(b).insertlast(a).deletelast.last)

= {by virtue of the second insertlast delete rule}

list(init.insertlast(b).last)

= {by virtue of the second last axiom}

b. QED

4. list(init.insertfirst(a).insertfirst(b).insertlast(c).insertat(d,3).deletefirst.deleteat(2).first) = **a**

= {by virtue of the first convert rule}

list(init.insertlast(b).insertfirst(a).insertlast(c).insertat(d,3).deletefirst.deleteat(2).first)

= {by virtue of the first convert rule}

list(init.insertlast(b).insertlast(a).insertlast(c).insertat(d,3).deletefirst.deleteat(2).first)

= {by virtue of the second convert rule}

list(init.insertlast(b).insertlast(a).insertlast(d).insertlast(c).deletefirst.deleteat(2).first)

= {by virtue of the first insertlast delete rule}

list(init.insertlast(a).insertlast(d).insertlast(c).deleteat(2).first)

= {by virtue of the third insertlast delete rule}

list(init.insertlast(a).insertlast(c).first)

= {by virtue of the second first axiom}

a. QED

5. list(init.insertfirst(a).insertlast(b).insertlast(c).insertlast(d).insertat(a,1).insertat(b,3).last)= **d**

= {by virtue of the first convert rule}

list(init.insertlast(a).insertlast(b).insertlast(c).insertlast(d).insertat(a,1).insertat(b,3).last)

= {by virtue of the second convert rule}

list(init.insertlast(a).insertlast(a).insertlast(b).insertlast(c).insertlast(d).insertat(b,3).last)

= {by virtue of the second convert rule}

list(init.insertlast(a).insertlast(a).insertlast(b).insertlast(b).insertlast(c).insertlast(d).last)

= {by virtue of the second last axiom}

d. QED

6. list(insertfirst(a).insertfirst(b).init.insertlast(c).deletefirst.empty) = **true**

= {by virtue of the init rule}

list(init.insertlast(c).deletefirst.empty)

= {by virtue of the first insertlast delete rule}

list(init.empty)

= {by virtue of the first empty axiom}

true. QED

7. list(init.insertfirst(a).init.insertlast(b).last.insertlast(c).deletelast.empty)= **false**

= {by virtue of the init rule}

list(init.insertlast(b).last.insertlast(c).deletelast.empty)

= {by virtue of the VX-op rules}

list(init.insertlast(b).insertlast(c).deletelast.empty)

= {by virtue of the second insertlast delete rule}

list(init.insertlast(b).empty)

= {by virtue of the second empty axiom}

false. QED

8. list(init.insertfirst(a).insertfirst(b).insertlast(c).insertlast(d).deleteat(3).first.insertat(a,2). search(c)) = **false**

= {by virtue of the VX-op rules}

list(init.insertfirst(a).insertfirst(b).insertlast(c).insertlast(d).deleteat(3).insertat(a,2). search(c))

= {by virtue of the first convert rule}

list(init.insertlast(b).insertfirst(a).insertlast(c).insertlast(d).deleteat(3).insertat(a,2). search(c))

= {by virtue of the first convert rule}

list(init.insertlast(b).insertlast(a).insertlast(c).insertlast(d).deleteat(3).insertat(a,2). search(c))

= {by virtue of the third insertlast delete rule}

list(init.insertlast(b).insertlast(a).insertlast(d).insertat(a,2).search(c))

= {by virtue of the third convert rule}

list(init.insertlast(b).insertlast(a).insertlast(a).insertlast(d).search(c))

= {by virtue of the search rule}

list(init.insertlast(b).insertlast(a).insertlast(a).search(c))

= {by virtue of the search rule}

list(init.insertlast(b).insertlast(a).search(c))

= {by virtue of the search rule}

list(init.insertlast(b).search(c))

= {by virtue of the search rule}

list(init.search(c))

= {by virtue of the search axiom}

false. QED

9. list(init.insertlast(a).insertfirst(a).insertlast(b).insertlast(c).deleteat(2).insertat(a,3).

multisearch(a)) = **2**

= {by virtue of the first convert rule}

list(init.insertlast(a).insertlast(a).insertlast(b).insertlast(c).deleteat(2).insertat(a,3).
multisearch(a))

= {by virtue of the third insertlast delete rule}

list(init.insertlast(a).insertlast(b).insertlast(c).insertat(a,3).multisearch(a))

= {by virtue of the third convert rule}

list(init.insertlast(a).insertlast(b).insertlast(a).insertlast(c).multisearch(a))

= {by virtue of the first multisearch rule, with h= < insertlast(a).insertlast(b)>}

1 + list(init.insertlast(a).insertlast(b).insertlast(c).multisearch(a))

= {by virtue of the first multisearch rule, with h= < >}

1 + 1 + list(init.insertlast(b).insertlast(c)multisearch(a))

= {by virtue of the second multisearch rule}

1 + 1 + list(init.insertlast(b).multisearch(a))

= {by virtue of the second multisearch rule}

1 + 1 + list(init.multisearch(a))

= {by virtue of the multisearch axiom}

1 + 1 + 0

= {arithmetic}

2. QED

10. list(init.insertfirst(a).init.insertlast(b).size.deletelast.insertlast(c).choose)= **c**

= {by virtue of the init rule}

list(init.insertlast(b).size.deletelast.insertlast(c).choose)

= {by virtue of the VX-op rules}

list(init.insertlast(b).deletelast.insertlast(c).choose)

= {by virtue of the second insertlast delete rule }

list(init.insertlast(c).choose)

= {by virtue of the second choose axiom, with h= < >}

c. QED

11. list(init.insertfirst(2).insertfirst(4).empty.insertlast(3).insertlast(2).deletefirst.deletelast.
smallest) =**2**

= {by virtue of the VX-op rules}

list(init.insertfirst(2).insertfirst(4).insertlast(3).insertlast(2).deletefirst.deletelast.smallest)

= {by virtue of the first convert rule}

list(init.insertlast(4).insertfirst(2).insertlast(3).insertlast(2).deletefirst.deletelast.smallest)

= {by virtue of the first convert rule}

list(init.insertlast(4).insertlast(2).insertlast(3).insertlast(2).deletefirst.deletelast.smallest)

= {by virtue of the first insertlast delete rule}

list(init.insertlast(2).insertlast(3).insertlast(2).deletelast.smallest)

= {by virtue of the second insertlast delete rule}

list(init.insertlast(2).insertlast(3).smallest)

= {by virtue of the smallest rule, with h=< insertlast(2) >}

Min (3, list(init.insertlast(2).smallest))

= {by virtue of the smallest rule, with h=< >}

Min (3 , 2 , list(init.smallest))

= {by virtue of the smallest axiom}

Min (3 , 2 , + ∞)

{arithmetic}

2. QED

12. list(init.insertfirst(4).insertfirst(7).insertlast(2).insertlast(1).deletefirst.largest) = **4**

= {by virtue of the first convert rule}

list(init.insertlast(7).insertfirst(4).insertlast(2).insertlast(1).deletefirst.largest)

= {by virtue of the first convert rule}

list(init.insertlast(7).insertlast(4).insertlast(2).insertlast(1).deletefirst.largest)

= {by virtue of the first insertlast delete rule}

list(init.insertlast(4).insertlast(2).insertlast(1).largest)

= {by virtue of the largest rule, with h = < insertlast(4).insertlast(2) >}

Max (1 , list(init.insertlast(4).insertlast(2).largest))

= {by virtue of the largest rule, with h = < insertlast(4) >}

Max (1 , 2, list(init.insertlast(4).largest))

= {by virtue of the largest rule, with h = < >}

Max (1 , 2, 4, list(init.largest))

= {by virtue of the largest axiom}

Max (1 , 2, 4, - ∞)

{arithmetic}

4. QED

APPENDIX -D

SPECIFYING ABSTRACT DATA TYPES IN THE SYNTAX OF ALNEELAIN SPECIFICATION LANGUAGE

1. QUEUE SPECIFICATION:

```
specification Queue;
    type
        itemtype : char;
    input
        vop front: itemtype ,
        vop rear: itemtype ,
        vop size: integer ,
        vop empty: Boolean
        oop init, dequeue, enqueue(itemtype)
    endinput;
    output
        itemtype ^ error ^  integer ^ Boolean
    endoutput;
    variable
        a: itemtype,
        b: itemtype,
        h: inputstar,
        hprime: inputstar,
        hplus: inputstar;
    axioms
        axiom frontAxioms:
            Queue(init.front)= error &
            Queue(init.enqueue(a).enqueue(b).front)= a,
        axiom rearAxioms:
            Queue(init.rear)= error &
            Queue(init.enqueue(b).enqueue(a).rear)= a,
        axiom sizeAxiom:
            Queue(init.size)= 0,
        axiom emptyAxioms:
            Queue(init.empty)= true &
            Queue(init.enqueue(a).empty)= false
    endaxioms;
    rules
        rule initRule:
            Queue(h.init.hprime) = Queue(init.hprime),
        rule initdequeueRule:
            Queue(init.dequeue.h) = Queue(init.h),
        rule enqueuedequeueRule:
            Queue(init.enqueue(a).enqueue(b).dequeue.hplus)=
            Queue(init.enqueue(b).hplus),
        rule sizeRule:
            Queue(init.h.enqueue(a).size) =1+ Queue(init.h.size),
        rule emptyRules:
```

```
                Queue(init.h.enqueue(a).hprime.empty)=>
                Queue(init.h.hprime.empty) &
                Queue(init.h.empty) => Queue(init.h.dequeue.empty),
        rule VopRules:
                Queue(init.h.front.hplus) = Queue(init.h.hplus) &
                Queue(init.h.rear.hplus) = Queue(init.h.hplus) &
                Queue(init.h.size.hplus) = Queue(init.h.hplus) &
                Queue(init.h.empty.hplus) = Queue(init.h.hplus)
    endrules;
endspecification
```

2. SEQUENCE SPECIFICATION:

```
specification Sequence;
    input
        vop head : char ,
        vop last: char ,
        vop length: integer,
        vop empty: Boolean
        oop init, deletehead, deletelast, puthead(char), putlast(char)
    endinput;
    output
        char ^ error ^  integer ^ Boolean
    endoutput;
    variable
        a: char ,
        b: char ,
        h: inputstar,
        hprime: inputstar,
        hplus: inputplus;
    axioms
        axiom LengthAxiom:
                Sequence(init.length) = 0,
        axiom emptyAxioms:
                Sequence(init.empty)= true &
                Sequence(init.puthead(a).empty)= false &
                Sequence(init.putlast(a).empty)= false ,
        axiom headAxioms:
                Sequence (init.head)= error &
                Sequence (init.h.putlast(b).puthead(a).head)= a &
                Sequence (init.h.puthead(b).puthead(a).head)= a &
                Sequence (init.h.puthead(a).putlast(b).head)= a ,
        axiom lastAxioms:
                Sequence (init.last)= error &
                Sequence (init.h.putlast(b).putlast(a).last)= a &
                Sequence (init.h.puthead(b).putlast(a).last)= a &
                Sequence (init.h.putlast(a).puthead(b).last)= a
    endaxioms;
    rules
        rule initRule:
                Sequence(h.init.hprime) = Sequence(init.hprime),
        rule initdeleteRules:
                Sequence(init.deletehead.h) = Sequence(init.h) &
                Sequence(init.deletelast.h) = Sequence(init.h) ,
```

rule putheaddeleteRules**:**
Sequence(init.h.puthead(a).deletehead.hplus)=Sequence(init.h.hplus)**&**
Sequence(init.puthead(a).deletelast.hplus) = Sequence(init.hplus) **,**
rule putlastdeleteRules**:**
Sequence(init.h.putlast(a).deletelast.hplus)= Sequence(init.h.hplus) **&**
Sequence(init.putlast(a).deletehead.hplus) = Sequence(init.hplus) **,**
rule lengthRule**:**
Sequence(init.h.putlast(a).length) =1+ Sequence(init.h.length) **,**
rule emptyRules**:**
Sequence(init.h.puthead(a).hprime.empty)=>
Sequence(init.h.hprime.empty) **&**
Sequence(init.h.putlast(a).hprime.empty)=>
Sequence(init.h.hprime.empty) **&**
Sequence(init.h.empty) => Sequence(init.h.deletehead.empty) **&**
Sequence(init.h.empty) => Sequence(init.h.deletelast.empty) **,**
rule VopRules**:**
Sequence(init.h.head.hplus) = Sequence(init.h.hplus) **&**
Sequence(init.h.last.hplus) = Sequence(init.h.hplus) **&**
Sequence(init.h.length.hplus) = Sequence(init.h.hplus) **&**
Sequence(init.h.empty.hplus) = Sequence(init.h.hplus)
endrules;
endspecification

3. <u>SET SPECIFICATION:</u>

specification Set;
type
itemtype: char;
input
vop choose : itemtype**,**
vop smallest: itemtype**,**
vop largest: itemtype**,**
vop List: itemtypestar,
vop size: integer **,**
vop search(itemtype): Boolean**,**
vop empty : Boolean
oop init, insert(itemtype), remove(itemtype)
endinput;
output
error ^ itemtype ^ plusinfinity ^ minusinfinity ^ emptyset ^ itemtypestar ^
integer ^ Boolean
endoutput;
variable
a: itemtype**,**
b: itemtype**,**
h: inputstar**,**
hprime: inputstar**,**
hplus: inputplus **;**
axioms
axiom sizeAxiom**:**
Set(init.size)= 0**,**
axiom emptyAxioms**:**
Set(init.empty)= true **&**

```
                Set(init.insert(a).empty)= false ,
        axiom searchAxiom:
                Set(init.search(a)) = false ,
        axiom chooseAxioms:
                Set(init.choose) = error &
                Set(init.h.insert(a).choose)=a ,
        axiom ListAxiom:
                Set(init.List) = emptyset ,
        axiom smallestAxiom :
                Set(init.smallest) = plusinfinity ,
        axiom largestAxiom:
                Set(init.largest) = minusinfinity
    endaxioms;
    rules
        rule initRule:
                Set(h.init.hprime) = Set(init.hprime) ,
        rule initremoveRule:
                Set(init.remove(a).h) = Set(init.h) ,
        rule insertremoveRule:
                Set(init.h.insert(a).remove(a).hplus) =  Set(init.h.hplus) ,
        rule sizeRule:
                IF(Set(init.h.search(a))) THEN
                Set(init.h.insert(a).size) = Set(init.h.size)
                ELSE Set(init.h.insert(a).size) = 1 + Set(init.h.size) ,
        rule searchRule:
                Set(init.h.insert(a).hprime.search(b))=
                (a=b) Set(init.h.hprime.search(a)) ,
        rule ListRule:
                Set(init.h.insert(a).List) = {a} ^ Set(init.h.List) ,
        rule commutativityRule:
                Set(init.h.insert(a).insert(b).hprime)=
                Set(init.h.insert(b).insert(a).hprime) ,
        rule insertRule:
                Set(init.h.insert(a).insert(a).hprime) = Set(init.h.insert(a).hprime) ,
        rule smallestRule:
                Set(init.h.insert(a).smallest) = MIN(a , Set(init.h.smallest)) ,
        rule largestRule:
                Set(init.h.insert(a).largest) = MAX(a , Set(init.h.largest)),
        rule emptyRules:
                Set(init.h.insert(a).hprime.empty)  => Set(init.h.hprime.empty)&
                Set(init.h.empty)  => Set(init.h. remove(a).empty) ,
        rule VopRules:
                Set(init.h.size.hplus) = Set(init.h.hplus) &
                Set(init.h.empty.hplus) = Set(init.h.hplus) &
                Set(init.h.search.hplus) = Set(init.h.hplus) &
                Set(init.h.choose.hplus) = Set(init.h.hplus) &
                Set(init.h.List.hplus) = Set(init.h.hplus) &
                Set(init.h.smallest.hplus) = Set(init.h.hplus) &
                Set(init.h.largest.hplus) = Set(init.h.hplus)
    endrules;
endspecification
```

4. MULTISET SPECIFICATION:

```
specification Multiset;
    input
        vop choose: char,
        vop least: char,
        vop most: char,
        vop smallest : char,
        vop largest : char,
        vop list : itemtypestar,
        vop lists(integer): itemtypestar,
        vop size : integer,
        vop multisize : integer,
        vop multisearch(char): integer,
        vop empty : Boolean,
        vop search(char): Boolean
        oop init, removeany, eraseany, insert(char), insertxn(char, integer),
        remove(char) , removexn(char, integer), removeall(char)
    endinput;
    output
        error ^ char ^ plusinfinity ^ minusinfinity ^ emptymultiset ^ itemtypestar ^
        integer ^ Boolean
    endoutput;
    variable
        a: char,
        b: char,
        n: integer,
        m: integer,
        h: inputstar,
        hprime: inputstar,
        hplus: inputplus;
    axioms
        axiom sizeAxiom:
            Multiset(init.size)= 0 ,
        axiom multisizeAxiom:
            Multiset(init.multisize)= 0,
        axiom emptyAxioms:
            Multiset(init.empty)= true &
            Multiset(init.insert(a).empty)= false,
        axiom searchAxiom:
            Multiset(init.search(a))= false ,
        axiom multisearchAxiom:
            Multiset(init.multisearch(a))= 0,
        axiom chooseAxioms:
            Multiset(init.choose) = error &
            Multiset(init.h.insert(a).choose)=a ,
        axiom listAxiom:
            Multiset(init.list) = emptymultiset,
        axiom listsAxiom:
            Multiset(init.lists(n)) = emptymultiset,
        axiom smallestAxiom:
            Multiset(init.smallest)= plusinfinity,
        axiom largestAxiom:
```

```
            Multiset(init.largest)= minusinfinity,
        axiom leastAxioms:
            Multiset (init.least)= error &
            Multiset (init.insert(a).insert(b).least)= (a ,b) ,
        axiom mostAxioms:
            Multiset (init.most)= error &
            Multiset (init.insert(a).insert(b).most)= (a , b)
endaxioms;
rules
        rule initRule:
            Multiset (h.init.hprime) = Multiset (init.hprime),
        rule initremoveRules:
            Multiset (init.remove(a).h) = Multiset (init.h) &
            Multiset (init.removexn(a, n).h) = Multiset (init.h) &
            Multiset (init.removeall(a).h) = Multiset (init.h) &
            Multiset (init.removeany.h) = Multiset (init.h) &
            Multiset (init.eraseany.h) = Multiset (init.h) ,
        rule insertremoveRules:
            Multiset(init.h.insert(a).remove(a).hplus) =  Multiset (init.h.hplus) &
            Multiset(init.h.insertxn(a, n).remove(a).hplus) =
            Multiset (init.h.insertxn(a,n-1).hplus) ,
        rule insertremovexnRule:
            IF (Multiset (init.h.multisearch(a)) <= n) THEN
            Multiset (init.h.insertxn(a, m).removexn(a, n). hplus) =
            Multiset (init.h.hplus)
            ELSE Multiset (init.h.insertxn(a, m).removexn(a, n).hplus) =
            Multiset (init.h. insertxn(a, m-n).hplus),
        rule insertremoveallRule:
            IF (Multiset (init.h.search(a))) THEN
            Multiset (init.h.insert(a).removeall(a).hplus) =  Multiset (init.h.hplus)
            ELSE
            Multiset (init.h.insert(b).removeall(a).hplus) =
            Multiset(init.h.insert(b).hplus) ,
        rule insertremoveanyRules:
           Multiset (init.h.insert(a).removeany.hplus) =  Multiset (init.h.hplus) &
           Multiset (init.h.insertxn(a, n).removeany.hplus) =
           Multiset (init.insertxn(a, n-1).h.hplus) ,
        rule inserteraseanyRule:
            Multiset (init.h.insert(a).eraseany.hplus) =  Multiset (init.h.hplus) ,
        rule sizeRule:
            IF (Multiset(init.h.search(a))) THEN
            Multiset(init.h.insert(a).size)=Multiset(init.h.size)
            ELSE Multiset(init.h.insert(a).size) = 1 + Multiset(init.h.size) ,
        rule multisizeRules:
            Multiset (init.h.insert(a).multisize) =  1+ Multiset (init.h.multisize)&
            Multiset (init.h.insertxn(a, n).multisize) = n +
            Multiset(init.h.multisize) ,
        rule searchRule:
            Multiset(init.h.insert(a).hprime.search(b))=
            (a=b)  Multiset(init.h.hprime.search(a)),
        rule multisearchRules:
            Multiset(init.h.insert(a).insert(b).multisearch(a)) = 1 +
            Multiset(init.h.insert(b).multisearch(a)) &
```

Multiset(init.h.insertxn(a, n).insertxn(b, n).multisearch(a)) = n +
Multiset(init.h.insertxn(b, n).multisearch(a)) **&**
Multiset(init.h.insert(b).multisearch(a)) =
Multiset(init.h.multisearch(a)) **,**

rule listRules**:**
Multiset (init.h.insert(a).list) = {a} ^ Multiset (init.h.list)**&**
Multiset (init.h.insertxn(a,n).list) = {a:n} ^ Multiset (init.h.list) **,**

rule listsRule**:**
IF (Multiset(init.h.multisearch(a)) >= n) **THEN**
Multiset (init.h.insert(a).lists(n)) = {a} ^ Multiset (init.h.lists(n))
ELSE Multiset (init.h.insert(a).lists(n)) = Multiset (init.h.lists(n)) **,**

rule commutativityRule**:**
Multiset(init.h.insert(a).insert(b).hprime)=
Multiset(init.h.insert(b).insert(a).hprime)**,**

rule smallestRule**:**
Multiset(init.h.insert(a).smallest) = MIN(a , Multiset(init.h.smallest))**,**

rule largestRule**:**
Multiset(init.h.insert(a).largest) = MAX(a , Multiset(init.h.largest)) **,**

rule leastRule**:**
IF (Multiset(init.h.multisearch(a)) < Multiset(init.h.multisearch(b)))
THEN
Multiset(init.h.insert(a).insert(b).least)= Multiset(init.h.insert(a).least)
ELSE
IF (Multiset(init.h.multisearch(a)) = Multiset(init.h.multisearch(b)))
THEN Multiset(init.h.insert(a).insert(b).least)=
Multiset (init.h.insert(a). insert(b).least)
ELSE
Multiset(init.h.insert(a).insert(b).least)= Multiset(init.h.insert(b).least)**,**

rule mostRule**:**
IF (Multiset(init.h.multisearch(a)) > Multiset(init.h.multisearch(b)))
THEN Multiset(init.h.insert(a).insert(b).most) =
Multiset (init.h.insert(a).most) **ELSE**
IF (Multiset(init.h.multisearch(a)) = Multiset(init.h.multisearch(b)))
THEN Multiset(init.h.insert(a).insert(b).most)=
Multiset(init.h.insert(a).insert(b).most)
ELSE
Multiset(init.h.insert(a).insert(b).most)=Multiset(init.h.insert(b).most) **,**

rule emptyRules**:**
Multiset (init.h.insert(a).hprime.empty) =>
Multiset (init.h.hprime.empty) **&**
Multiset (init.h.empty) => Multiset (init.h.remove(a).empty) **&**
Multiset (init.h.empty) => Multiset (init.h. removexn(a, n).empty) **&**
Multiset (init.h.empty) => Multiset (init.h. removeall(a).empty) **&**
Multiset (init.h.empty) => Multiset (init.h. removeany.empty) **&**
Multiset (init.h.empty) => Multiset (init.h. eraseany.empty) **,**

rule VopRules**:**
Multiset (init.h.size.hplus) = Multiset (init.h.hplus) **&**
Multiset (init.h.multisize.hplus) = Multiset (init.h.hplus) **&**
Multiset (init.h.empty.hplus) = Multiset (init.h.hplus) **&**
Multiset (init.h.search(a).hplus) = Multiset (init.h.hplus) **&**
Multiset (init.h.multisearch(a).hplus) = Multiset (init.h.hplus) **&**
Multiset (init.h.choose.hplus) = Multiset (init.h.hplus) **&**
Multiset (init.h.list.hplus) = Multiset (init.h.hplus) **&**
Multiset (init.h.lists(n).hplus) = Multiset (init.h.hplus) **&**

```
                    Multiset (init.h.least.hplus) = Multiset (init.h.hplus) &
                    Multiset (init.h.most.hplus) = Multiset (init.h.hplus) &
                    Multiset (init.h.smallest.hplus) = Multiset (init.h.hplus) &
                    Multiset (init.h.largest.hplus) = Multiset (init.h.hplus)
        endrules;
endspecification
```

5. LIST SPECIFICATION:

```
specification List;
        type
                itemtype: char;
        input
                vop choose : itemtype,
                vop first : itemtype,
                vop last: itemtype,
                vop smallest: itemtype,
                vop largest: itemtype,
                vop size : integer,
                vop multisearch(itemtype): integer,
                vop empty : Boolean,
                vop search(itemtype): Boolean
                oop init, deletelast, deletefirst, deleteat(integer), insertlast(itemtype) ,
                insertfirst(itemtype) , insertat(itemtype, integer)
        endinput;
        output
                error ^ itemtype ^ plusinfinity ^ minusinfinity ^ integer ^ Boolean
        endoutput;
        variable
                a: itemtype,
                b: itemtype,
                n: integer,
                h: inputstar,
                hprime: inputstar,
                hplus: inputplus;
        axioms
                axiom sizeAxiom:
                      List(init.size)= 0,
                axiom emptyAxioms:
                        List(init.empty)= true &
                        List(init.insertlast(a).empty)= false,
                axiom searchAxiom:
                        List(init.search(a))= false ,
                axiom multisearchAxiom:
                        List(init.multisearch(a))= 0,
                axiom chooseAxioms:
                        List(init.choose) = error &
                        List(init.h.insertlast(a).choose)=a,
                axiom firstAxioms:
                        List(init.first)= error &
                        List(init.insertlast(a).insertlast(b).first)= a ,
                axiom lastAxioms:
                        List(init.last)= error &
                        List(init.insertlast(b) .insertlast(a).last)= a,
```

```
        axiom smallestAxiom:
                List(init.smallest) = plusinfinity,
        axiom largestAxiom:
                List(init.largest) = minusinfinity
endaxioms;
rules
        rule initRule:
                List(h.init.hprime) = List(init.hprime) ,
        rule initdeleteRules:
                List(init.deletefirst.h) = List(init.h)&
                List(init.deletelast.h) = List(init.h)&
                List(init.deleteat(n).h) = List(init.h) ,
        rule insertlastdeleteRules:
                List(init.insertlast(a).insertlast(b).deletefirst.hplus) =
                List(init.insertlast(b).hplus)&
                List(init.insertlast(b). insertlast(a).deletelast.hplus) =
                List(init.insertlast(b) .hplus)&
                IF (List(init.h.hprime.size) >= n ) THEN
                List(init.h.insertlast(a).deleteat(n).hprime) = List(init.h.hprime)
                ELSE  List(init.h.insertlast(a).deleteat(n).hprime) =
                List(init.h.insertlast(a).hprime) ,
        rule searchRule:
                List(init.h.insertlast(a).hprime.search(b)) =
                (a=b) List(init.h.hprime.search(a)) ,
        rule multisearchRules:
                List(init.h.insertlast(a).insertlast(b).multisearch(a)) = 1 +
                List(init.h.insertlast(b).multisearch(a))&
                List(init.h.insertlast(b).hprime.multisearch(a))=
                List(init.h.hprime.multisearch(a)) ,
        rule sizeRule:
                List(init.h.insertlast(a).size) =  1+ List(init.h.size) ,
        rule convertRules:
                List(init.h.insertfirst(a).hprime) = List(init.h.insertlast(a).hprime)&
                IF ( List(init.h.hprime.size) < n ) THEN
                List(init.h.insertat(a, n).hprime) = List(init.h.hprime)
                ELSE List(init.h.insertat(a, n).hprime) =
                List(init.h.insertlast(a).hprime) ,
        rule smallestRule:
                List(init.h.insertlast(a).smallest) = MIN(a , List(init.h.smallest)),
        rule largestRule:
                List(init.h.insertlast(a).largest) = MAX(a , List(init.h.largest)),
        rule emptyRules:
               List(init.h.insertlast(a).hprime.empty)  => List(init.h.hprime.empty) &
                List(init.h.empty)  => List(init.h. deletefirst.empty) &
                List(init.h.empty)  => List(init.h. deletelast.empty) &
                List(init.h.empty)  => List(init.h. deleteat(n).empty) ,
        rule VopRules:
                List(init.h.size.hplus) = List(init.h.hplus)&
                List(init.h.empty.hplus) = List(init.h.hplus) &
                List(init.h.search(a).hplus) = List(init.h.hplus) &
                List(init.h.multisearch(a).hplus) = List(init.h.hplus) &
                List(init.h.choose.hplus) = List(init.h.hplus) &
                List(init.h.first.hplus) = List(init.h.hplus) &
                List(init.h.last.hplus) = List(init.h.hplus) &
```

```
                    List(init.h.smallest.hplus) = List(init.h.hplus) &
                    List(init.h.largest.hplus) = List(init.h.hplus)
      endrules;
endspecification
```

APPENDIX -E

AN EXAMPLE FOR VERIFYING QUEUE DATA TYPE IMPLEMENTATION

1. AXIOMS OF QUEUE DATA TYPE

- **Size Axiom**

  Queue(init.size) = 0.

- **Init Empty Axiom**

  Queue(init.empty) = true.

- **Enqueue Empty Axiom**

  Queue(init.enqueue(a).empty) = false.

- **Init Front Axiom**

  Queue(init.front) = error.

- **Enqueue Front Axiom**

  Queue(init.enqueue(a).enqueue(b).front) = a.

- **Init Rear Axiom**

  Queue(init.rear) = error.

- **Enqueue Rear Axiom**

  Queue(init.enqueue(b).enqueue(a).rear) = a.

2. AN IMPLEMENTATION OF QUEUE DATA TYPE

This source code is taken from the Apache C++ Standard Library (STDCXX)).

```
//************************************************************
//  Header file queue.h
//************************************************************
const int SIZE = 100;
// an approximation of infinity: unbounded queue

typedef int qitemtype;
typedef int indextype;
```

```
class queue

   { public:

    queue();                // default constructor

    void init();       // initializes or re-initializes the queue

    bool empty () const;   // tells whether queue is empty

    void enqueue (qitemtype qitem); //add qitem to the end of the queue

    void dequeue (); //deletes the element at the front of the queue

    qitemtype front (); // returns the element at the front of the queue

    qitemtype rear (); // returns the element at the end of the queue

    int size ();                // returns size of the queue

    private:
    // array-based implementation.

    qitemtype qarray [SIZE];
    indextype qindex;
    int front1,rear1;
   };

// **************************************************************
// Array-based C++ implementation for the queue ADT.
// file queue.cpp, refers to header file queue.h.
// **************************************************************

#include "queue.h" // queue.h header file.

queue::queue()
{
    front1=-1;
    rear1=-1;
    qindex=0;
}

bool queue :: empty ()  const
{
    return (qindex==0);
}

void queue :: init ()
{
     qindex =0;
     front1=-1;
     rear1=-1;

}

void queue::enqueue(int qitem)
{
    if(front1==-1)
    {
        front1= front1+1;
```

```
        rear1= rear1+1;
        qarray[rear1]=qitem;
        qindex=qindex+1;
    }
    else if(rear1>=SIZE-1)
    {
        return;
     }
    else
    {
        rear1=rear1+1;
        qarray[rear1]=qitem;
        qindex=qindex+1;
    }
}

void queue::dequeue()
{
     if(front1==-1)
     {
          cout<<"Deletion is not possible:: Queue is empty
\n"<<endl;
          return;
     }
     else
     {
          if(front1==rear1)
          {
                front1=rear1=-1;
                qindex=0;
                return;
          }
          cout<<"The deleted element is
\n"<<qarray[front1]<<endl;
          front1=front1+1;
          qindex=qindex-1  ;
     }
}

qitemtype queue :: front ()
{
    int error = -9999;
    if (front1>=0)
    {
        return qarray[front1];
    }
    else
    {
        return error;
    }
}

qitemtype queue :: rear ()
{
    int error = -9999;
    if (rear1>=0)
    {
        return qarray[rear1];
    }
    else
```

```
    {
        return error;
    }
}

int queue :: size ()
{
    return qindex;
}
```

3. MAPPING AXIOMS TO HOARE FORMULAS

- **Size Axiom**

  *v: {**true**} init(); y=size() {y=0}*

  Where y is a variable of type integer.

- **Init Empty Axiom**

  *v: {**true**} init(); y=empty() {y=true}*

  Where y is a variable of type Boolean.

- **Enqueue Empty Axiom**

  *v:{**true**} init();enqueue(a);y=empty();{y= false}*

  For an arbitrary item a, where y is a variable of type Boolean.

- **Init Front Axiom**

  *v: {**true**} init(); y=front() {y=error}*

  Where y is declared as a variable that can hold the data type of the items that we put on the queue (called itemtype) or the error message.

- **Enqueue Front Axiom**

  *v:{**true**} init();enqueue(a); enqueue(b);y=front();{y= a}*

  For an arbitrary item a, where y is a variable of type itemtype and *h* is an arbitrary sequence of operations.

- **Init Rear Axiom**

  *v: {**true**} init(); y=rear() {y=error}*

  Where y is declared as a variable that can hold the data type of the items that we put on the queue (called itemtype) or the error message.

- **Enqueue Rear Axiom**

  *v:{**true**} init();enqueue(b); enqueue(a);y=rear();{y= a}*

  For an arbitrary item a, where y is a variable of type itemtype and *h* is an arbitrary sequence of operations.

4. VERIFYING QUEUE IMPLEMENTATION AGAINST AXIOMS

1. **Size Axiom:** *v: {**true**} init(); y=size() {y=0}*

v: {true}

qindex :=0;

front1:=-1;

rear1:=-1;

y:= (return qindex)

{y=0}

In order to prove formula v, we apply the sequence rule and generate the following formulas using *{qindex=0}* as intermediate assertion:

v0: {true} qindex:=0 {qindex=0}

v1: {qindex=0} front1:= -1 {qindex=0}

v2: {qindex=0} rear1:= -1 {qindex=0}

v3: {qindex=0} y:= qindex{y=0}

By applying the assignment statement rule to v0, v1, v2 and v3 we find:

v00: true $\Rightarrow$ 0=0,

v10: qindex=0 $\Rightarrow$ qindex =0

v20: qindex=0 $\Rightarrow$ qindex =0

v30: qindex=0 $\Rightarrow$ qindex =0

All of which are tautologies, hence axioms of Hoare logic. This concludes the proof, and establishes the validity of v.

2. **Init Empty Axiom:** *v: {**true**} init(); y=empty() {y=true}*

v: {true}

qindex :=0;

front1:=-1;

rear1:=-1;

y:=(qindex=0)

{y=true}

In order to prove formula v, we apply the sequence rule and generate the following formulas using *{qindex=0}* as intermediate assertion:

v0: {true} qindex: =0 {qindex=0}

v1: {qindex=0} front1:=-1 {qindex=0}

v2: {qindex=0} rear1:=-1 {qindex=0}

v3: {qindex=0} y :=(qindex=0) {y=true}

By applying the assignment statement rule to v0, v1, v2 and v3 we find:

v00: true ⇒ 0=0,

v10: qindex=0 ⇒ qindex=0,

v20: qindex=0 ⇒ qindex=0,

v30: qindex=0 ⇒ (qindex=0) =true,

All of which are tautologies, hence axioms of Hoare logic. This concludes the proof, and establishes the validity of v.

3. **Init Front Axiom:** *v: {**true**} init(); y=front() {y=error}*

v: {true}

qindex:=0;

front1:=-1;

rear1:=-1;

if (front1>=0){y:= qarray[front1];} else {return error;}

{y=error}

In order to prove formula v, we apply the sequence rule and generate the following formulas using *{qindex=0}* and {qindex=0 ∧ front1= -1} as intermediate assertion:

v0: {true} qindex:=0 {qindex=0}.

v1: {qindex=0} front1:=-1 {qindex=0 ∧ front1= -1}.

v2: {qindex=0 ∧ front1= -1} rear1:=-1 {qindex=0 ∧ front1= -1}.

v3: {qindex=0 ∧ front1=-1} {y:= qarray[front1];}else {return error;} {y=error }

By applying the assignment statement rule to v0, v1 and v2 we find:

v00: true ⇒ 0=0

v10: qindex=0 ⇒ qindex=0 ∧ front1= -1,

v20: qindex=0 ∧ front1= -1 ⇒ qindex=0 ∧ front1= -1,

All of which are tautologies, hence axioms of Hoare logic. We consider v3, to which we apply the alternation rule. We find,

v30: {qindex=0 ∧ front1= -1 ∧ front1>=0} y: = qarray[front1]; {y=error}

v31: {qindex=0 ∧ front1= -1 ∧ front1<0} y:= error; {y=error}

The formula v30 is vacuously valid since the precondition is false. We apply the assignment statement rule to v31:

v310: qindex=0 ∧ front1<0 ⇒ error = error

This formula is a tautology, hence an axiom of Hoare's inference system; this concludes our proof.

4. **Init Rear Axiom:** *v: {**true**} init(); y=rear() {y=error}*

v: {true}

qindex :=0;

front1:=-1;

rear1:=-1;

if (rear1>=0){y:=qarray[rear1];} else{y:= error;}

{y=error}

In order to prove formula v, we apply the sequence rule and generate the following formulas using *{qindex=0}* and {qindex=0 ∧ rear1=-1} as intermediate assertions:

v0: {true} qindex:=0 {qindex=0}.

v1: {qindex=0} front1:=-1{qindex=0}.

v2: {qindex=0} rear1:=-1 {qindex=0 ∧ rear1=-1},

v3: {qindex=0 ∧ rear1=-1} if (rear1>=0){y:=qarray[rear1];}else{y:= error;} {y=error }.

By applying the assignment statement rule to v0, v1 and v2 we find:

v00: true ⇒ 0=0

v10: qindex=0 ⇒ qindex=0,

v20: qindex=0 ⇒ qindex=0 ∧ rear1= -1,

All of which are tautologies, hence axioms of Hoare logic. We consider v3, to which we apply the alternation rule. We find,

v30: {qindex=0 ∧ rear1=-1 ∧ rear1>=0} y: = qarray[rear1]; {y=error}

v31: {qindex=0 ∧ rear1=-1 ∧ rear1<0} y: = error; {y=error}

The formula v30 is vacuously valid since the precondition is false. We apply the assignment statement rule to v31:

v310: qindex=0 ∧ rear1<0 ⇒ error = error

This formula is a tautology, hence an axiom of Hoare's inference system; this concludes our proof.

5. SUBMITTING TO HAHA THROUGH THE API

1. **Size Axiom:** *v: {**true**} init(); y=size() {y=0}*
   v: {true}

qindex :=0;

front1:=-1;

rear1:=-1;

y:= qindex;

{y=0}

SizeAxiom.haha:

```
function S() : Z
  var
    qindex : Z
    front1: Z
    rear1: Z
    y : Z
begin
  skip
  {true}
  qindex:=0
  {qindex = 0}
  front1:=-1
  {qindex = 0}
  rear1:=-1
  {qindex = 0}
  y:=qindex
  { y = 0 }
  skip
end
```

2. **Init Empty Axiom:** *v: {**true**} init(); y=empty() {y=true}*

   v: {true}

qindex :=0;

front1:=-1;

rear1:=-1;

y:=(qindex=0)

{y=true}

InitEmptyAxiom.haha:

```
function S() : Z
```

```
  var
    qindex : Z
    front1: Z
    rear1: Z
    y : Z
begin
  skip
  {true}
  qindex:=0
  {qindex = 0}
  front1:=-1
  {qindex = 0}
  rear1:=-1
  {qindex = 0}
  if (qindex=0) then
  y:=1
  else
  y:=0
  { y= 1 }
  skip
end
```

3. **Enqueue Empty Axiom:** v*: {**true**} init(); enqueue(a); y=empty() {y=false}*

v: {true}

```
qindex :=0;
front1:=-1;
rear1:=-1;
if(front1==-1){
    front1= front1+1;
    rear1= rear1+1;
    qarray[rear1]=qitem;
    qindex=qindex+1;}
    else if(rear1>=SIZE-1)
    { return; }
    else {
        rear1=rear1+1;
        qarray[rear1]=qitem;
        qindex=qindex+1; }
}
y:=(sindex=0)
```

{y=false}

<u>EnqueueEmptyAxiom.haha:</u>

```
function S( ) : Z
var qindex : Z
      front1 : Z
      rear1  : Z
      qarray : ARRAY[Z]
      y      : Z
      qitem: Z
      SIZE: Z
begin
skip
{true}
  qindex := 0
  { qindex = 0 }
  front1 := -1
  { qindex = 0 and front1 = -1 }
  rear1 := -1
  { qindex = 0 and front1 = -1 and rear1 = -1 }
  if (front1 = -1) then
  begin
      front1 := front1 + 1
      {qindex = 0 and front1 = 0 and rear1 = -1  }
      rear1 := rear1+1
      {qindex = 0 and front1 = 0 and rear1 = 0 }
      qarray[rear1] := qitem
      {qindex=0 and front1=0 and rear1=0 and qarray[0] = qitem}
      qindex := qindex+1
  end
  else begin
    if rear1>=SIZE-1 then
      begin

      end
    else
      begin
        rear1 := rear1+1
        { qindex = 0 and front1 <> -1 and rear1 = 0  }
        qarray[rear1] := qitem
      {qindex=0 and front1<>-1 and rear1=0 and qarray[0]=qitem}
        qindex := qindex+1
      end
    end
  {qindex=1 and front1<>-1 and rear1=0 and qarray[0]= qitem}
      if (qindex=0)  then
        y := 1
      else
```

```
        begin
          y := 0
          { y= 0 }
          skip
        end
  skip
end
```

4. Init Front Axiom: *v: {**true**} init(); y=front() {y=error}*

v: {true}

qindex :=0;

front1:=-1;

rear1:=-1;

if (front1>=0){y:= qarray[front1];} else {return error;}

{y=error}

<u>**InitFrontAxiom.haha:**</u>

```
function S() : Z
  var
    qindex : Z
    front1: Z
    rear1: Z
    qarray: ARRAY[Z]
    y : Z
    error: Z
begin
  skip
  {true}
  qindex:=0
  {qindex = 0}
  front1:=-1
  {qindex = 0 and front1=-1}
  rear1:=-1
  {qindex = 0 and front1=-1 }
  if (front1>=0) then
     y := qarray[front1]
  else
     y:= error
  { y = error }
  skip
end
```

5. **Enqueue Front Axiom:** *v:{**true**} init();enqueue(a); enqueue(b);y=front();{y= a}*

v: {true}

```
qindex :=0;
front1:=-1;
rear1:=-1;
if(front1==-1){
    front1= front1+1;
    rear1= rear1+1;
    qarray[rear1]=qitem;
    qindex=qindex+1;}
    else if(rear1>=SIZE-1)
    { return; }
    else {
        rear1=rear1+1;
        qarray[rear1]=qitem;
        qindex=qindex+1; }
        }
if(front1==-1){
    front1= front1+1;
    rear1= rear1+1;
    qarray[rear1]=qitem;
    qindex=qindex+1;}
    else if(rear1>=SIZE-1)
    { return; }
    else {
        rear1=rear1+1;
        qarray[rear1]=qitem;
        qindex=qindex+1; }
        }
if (front1>=0){y:= qarray[front1];} else {return error;}
```

{y=a}

EnqueueFrontAxiom.haha:

```
function S( ) : Z
var qindex : Z
      front1 : Z
      rear1  : Z
      qarray : ARRAY[Z]
      y      : Z
      a: Z
      b: Z
      SIZE: Z
      error: Z
begin
skip
{true}
  qindex := 0
  { qindex = 0 }
  front1 := -1
  { qindex = 0 and front1 = -1 }
  rear1 := -1
  { qindex = 0 and front1 = -1 and rear1 = -1 }
  if (front1 = -1) then
  begin
      front1 := front1 + 1
      { qindex = 0 and front1 = 0 and rear1 = -1  }
      rear1 := rear1+1
      { qindex = 0 and front1 = 0 and rear1 = 0 }
      qarray[rear1] := a
      {qindex =0 and front1 =0 and rear1 =0 and qarray[0] = a }
      qindex := qindex+1
  end
  else begin
    if rear1>=SIZE-1 then
      begin

      end
    else
      begin
        rear1 := rear1+1
        { qindex = 0 and front1 = 0 and rear1 = 0  }
        qarray[rear1] := a
        {qindex=0 and front1=0 and rear1=0 and qarray[0] = a }
        qindex := qindex+1
      end
    end
  { qindex = 1 and front1 = 0 and rear1 = 0 and qarray[0] =a }
  if (front1 = -1) then
  begin
      front1 := front1 + 1
```

```
      {qindex=1 and front1=0 and rear1=0 and qarray[0]= a  }
      rear1 := rear1+1
      {qindex =1 and front1 =0 and rear1 =1 and qarray[0] =a }
      qarray[rear1] := b
      {qindex =1 and front1 =0 and rear1 =1  and qarray[0]=a
       and qarray[1] = b }
      qindex := qindex+1
  end
  else begin
    if rear1>=SIZE-1 then
      begin

      end
    else
      begin
        rear1 := rear1+1
        {qindex =1 and front1 =0 and rear1 =1 and qarray[0] =a}
        qarray[rear1] := b
        {qindex =1 and front1 =0 and rear1 =1 and qarray[0]=a
         and qarray[1] = b }
        qindex := qindex+1
      end
    end
     {qindex =2 and front1 =0 and rear1 =1 and qarray[0] =a
      and qarray[1] = b  }
      if (front1>=0) then
      y:= qarray[front1]
      else
      y:= error
      { y= a }
          skip
        end
```

6. Init Rear Axiom: *v: {**true**} init(); y=rear() {y=error}*

v: {true}

qindex :=0;

front1:=-1;

rear1:=-1;

if (rear1>=0){y:=qarray[rear1];} else{y:= error;}

{y=error}

<u>InitRearAxiom.haha:</u>

```
function S() : Z
  var
    qindex : Z
    front1: Z
    rear1: Z
    qarray: ARRAY[Z]
    y : Z
    error: Z
begin
  skip
  {true}
  qindex:=0
  {qindex = 0}
  front1:=-1
  {qindex = 0 and front1=-1}
  rear1:=-1
  {qindex = 0 and front1=-1 and rear1=-1}
  if (rear1>=0) then
     y := qarray[rear1]
  else
     y:= error
  { y = error }
  skip
end
```

7. Enqueue Rear Axiom: *v:{**true**} init();enqueue(b); enqueue(a);y=rear();{y= a}*

v: {true}

```
qindex :=0;
front1:=-1;
rear1:=-1;
if(front1==-1){
   front1= front1+1;
   rear1= rear1+1;
   qarray[rear1]=qitem;
   qindex=qindex+1;}
   else if(rear1>=SIZE-1)
   { return; }
   else {
      rear1=rear1+1;
```

```
                qarray[rear1]=qitem;
                qindex=qindex+1; }
                }
            if(front1==-1){
              front1= front1+1;
              rear1= rear1+1;
              qarray[rear1]=qitem;
              qindex=qindex+1;}
              else if(rear1>=SIZE-1)
              { return; }
              else {
                rear1=rear1+1;
                qarray[rear1]=qitem;
                qindex=qindex+1; }
                }
    if (rear1>=0){y:=qarray[rear1];} else{y:= error;}
                                                        {y=a}
```

EnqueueRearAxiom.haha:

```
function S( ) : Z
var qindex : Z
      front1 : Z
      rear1  : Z
      qarray : ARRAY[Z]
      y      : Z
      a: Z
      b: Z
      SIZE: Z
      error: Z
begin
skip
{true}
  qindex := 0
  { qindex = 0 }
  front1 := -1
  { qindex = 0 and front1 = -1 }
  rear1 := -1
  { qindex = 0 and front1 = -1 and rear1 = -1 }
  if (front1 = -1) then
  begin
      front1 := front1 + 1
```

```
    { qindex = 0 and front1 = 0 and rear1 = -1  }
    rear1 := rear1+1
    { qindex = 0 and front1 = 0 and rear1 = 0 }
    qarray[rear1] := b
    { qindex =0 and front1 =0 and rear1 =0 and  qarray[0] =b}
    qindex := qindex+1
end
else begin
  if rear1>=SIZE-1 then
    begin

    end
  else
    begin
      rear1 := rear1+1
      { qindex = 0 and front1 = 0 and rear1 = 0  }
      qarray[rear1] := b
      {qindex =0 and front1 =0 and rear1 =0 and qarray[0]=b }
      qindex := qindex+1
    end
  end
{qindex = 1 and front1 = 0 and rear1 = 0 and qarray[0] = b  }
if (front1 = -1) then
begin
    front1 := front1 + 1
    {qindex =1 and front1 =0 and rear1 =0 and qarray[0] =b}
    rear1 := rear1+1
    {qindex =1 and front1 =0 and rear1 =1 and qarray[0] =b}
    qarray[rear1] := a
    { qindex =1 and front1 =0 and rear1 =1  and qarray[0] =b
      and qarray[1] = a }
    qindex := qindex+1
end
else begin
  if rear1>=SIZE-1 then
    begin

    end
  else
    begin
      rear1 := rear1+1
      {qindex =1 and front1 =0 and rear1 =1 and qarray[0] =b}
      qarray[rear1] := a
      {qindex =1 and front1 =0 and rear1 =1 and qarray[0]= b
       and qarray[1] = a }
      qindex := qindex+1
    end
  end
```

```
{qindex =2 and front1 =0 and rear1 =1 and qarray[0] =b
  and qarray[1] = a  }
 if (rear1>=0) then
 y:= qarray[rear1]
 else
 y:= error
 { y= a }
     skip
   end
```