كلية الدراسات العليا

Sudan University of Science and Technology

College of Graduate Studies

An Automated Technique to Support Generation and Validation of Software Specifications

طريقة آلية لدعم إنشاء مواصفات البرمجيات والتحقق منها

A thesis Submitted for the Degree of Doctor of Philosophy in Computer Science

By

ABDELRASOUL YAHYA IBRAHIM MOSA

B.Sc. in Statics and Computer Science, University of Gezira

M.Sc. in Computer Science, University of Gezira

Supervisor: Prof. Dr. Ali Mili

September, 2018

Declaration

I hereby declare that this thesis is the result of my own investigation, except where otherwise stated. I also declare that it has not been previously or concurrently submitted as a whole for any other degrees at Sudan University of Science and Technology or other institutions.

Abdelrasoul Yahya Ibrahim Mosa

Signature________________ Date___________________

INTRODUCTIVE PAGE

قال تعالى:

(وَآتَاكُم مِّن كُلِّ مَا سَأَلْتُمُوهُ وَإِن تَعُدُّوا نِعْمَتَ اللَّهِ لَا تُحْصُوهَا إِنَّ الْإِنسَانَ لَظَلُومٌ كَفَّار**)**

الآية (34) من سورة ابراهيم

DEDICATION

I dedicate this project to God Almighty my creator, my source of inspiration, knowledge and understanding. He has been the source of my strength throughout this program and on His wings only have I soared. I also dedicate this work to my parent who always loved me unconditionally and they have continually prayed for my success. This thesis work is dedicated to my wife, who has been a constant source of support and encouragement during the study. Thank you for you all can never be quantified. God bless you.

ACKNOWLEDGEMENTS

My deepest gratitude goes to God who has provided all that was needed to complete this project and the program for which it was undertaken for. Throughout this entire study, He took care of everything that would have stopped me in my tracks and strengthened me even through my most difficult times.

I sincere appreciation also goes to my supervisor Professor Ali Mili whose contribution and constructive criticism has pushed me to expend the kind effort I have exerted to make this work as original as it can be. Thanks to him I have experienced true research and my knowledge on the subject matter has been broadened. I will never forget you sir.

I owe my deepest gratitude to my colleagues Nahid A.Ali and Amal A. Mirghani who are participated with me in the same project, and we are worked as team in the most of the study. I wish to thank them for their support and the nice time we have spent together.

ABSTRACT

This study presents a new technique to support software specification and validation. Such validation by means validate requirement specification itself prior to proceeding the next phase of software engineering lifecycle. The study based on the data types 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; specification is represented by axioms and rules, where axioms reflect the behavioral of the data type for simple data values and rules define their behavioral inductively for more complex data values. These axiomatic specifications are validated for completeness against independently generated data. Our challenge is to ensure that the specifications of abstract data types are valid and reflect all the relevant requirements and constraints for the completeness, consistency, and minimality because any faults that arise in this phase it might cause a negative impact on all subsequent phases and increase the cost of a software product. The methodology that we are follows is defined and specified abstract data types by using axiomatic representation, specified and checked them by using proposed language, and validated generated specification by using proposed validation tool. In order to satisfy the important criteria of a software specification such as simplicity, formality, and abstraction we have proposed a behavioral model that is used to specify abstract data types. This model is as follows: 1) an input space which defines a set of input space signals that abstract data type can accept; which his has two types: a) v operations that are expected to return or produce a value; b) o operations which are do not return or produce a value, but can affect future behavior of abstract data type. The relation between an input space and an output space is defines the name and the type of the abstract data type. Alneelain Specification Language has been proposed to implement the proposed model. Two components of its compiler have been developed; lexical analyser and syntax checker. Six samples of abstract data types have been selected as examples specified by Alneelain Specification Language, and checked its compiler. Alneelain Validation Tool which relied on the output of Alneelain Specification Language has been developed to use to validate completely independent validation data. The results showed the possibility of using Alneelain specification language to specify abstract data types and the possibility of using Alneelain Validation Tool to validate the specification of those abstract data types.

مستخلص البحث

تقدم هذه الدراسة تقنية جديدة لدعم مواصفات البرامج والتحقق منها. هذا التحقق يتم عبر وسائل التحقق من صحة المتطلبات نفسها قبل البدء في المرحلة التالية من دورة حياة هندسة البرمجيات. تعتمد الدراسة على أنواع البيانات من خلال مواصفات سلوكية تصف الخصائص الوظيفية لأنواع البيانات، ولكنها لا تعطي فكرة عن الكيفية التي يمكن بها تنفيذ أنواع البيانات؛و يتم تمثيل المواصفات باستخدام بديهيات وقواعد، حيث تعكس البديهيات سلوك أنواع البيانات لقيم بيانات بسيطة أما القواعد فانها تحدد سلوكا رسمياً لأنواع البيانات لقيم بيانات أكثر تعقيداً. يتم التحقق من هذه المواصفات البديهية للتأكد من أنها مواصفات مكتملة بواسطة بيانات تم إنشاؤها بطريقة مستقلة. ويتمثل التحدي الذي يواجهنا في التأكد من أن مواصفات تلك أنواع البيانات المجردة صحيحة وأنها تعكس جميع المتطلبات والقيود ذات الصلة بالاكتمال ، الاتساق والحد الأدنى ، لأن أي أخطاء تنشأ في هذه المرحلة قد تسبب أثراً سلبياً على جميع المراحل اللاحقة وزيادة في تكلفة منتج البرمجيات. وتحدد منهجيتنا التي اتبعناها تعريف وتوصيف أنواع البيانات المجردة باستخدام التمثيل البديهي ، وثم توصيفها باستخدام لغة مواصفات مقترحة والتأكد من صحة ذلك التوصيف ، وأخيراً التحقق من صحته باستخدام أداة مقترحة. ومن أجل تحقيق معايير مواصفات البرمجيات الهامة مثل البساطة، والرسمية والتجريد فقد اقترحنا نموذج سلوكي يستخدم لتوصيف لأنواع البيانات المجردة وهو كما يلي: 1) فضاء مدخلات والذي يحدد مجموعة من إشارات مدخلات يمكن لأنواع البيانات المجردة أن تقبلها وهي نوعان: أ) عمليات V وهي عمليات متوقع أن ترجع أو تنتج قيمة وب) عمليات O وهي عمليات لا ترجع أو تنتج قيمة ولكنها تؤثر في السلوك المستقبلي لأنواع البيانات المجردة. 2) فضاء مخرجات والذي يعرف مجموعة من قيم المخرجات والتي يمكن أن تنتجها أو ترجعها أنواع البيانات المجردة، والعلاقة بين فضاء المدخلات وفضاء المخرجات يعرفان اسم ونوع أنواع البيانات المجردة. تم اقتراح لغة مواصفات النيلين لتطبيق النموذج المقترح. وقد تم تطوير مترجم لهذه اللغة يحتوي على جزئين من مكوناته هما محلل المفردات ومدقق بناء الجملة . تم اختيار ستة عينات من أنواع البيانات المجردة كأمثلة برمجية وتم توصيفها بلغة النيلين للمواصفات وتم فحصها واختبارها بواسطة مترجم اللغة. تم تطوير أداة النيلين للتحقق من صحة المواصفات والتي تعتمد في تكوينها على مخرجات لغة النيلين للمواصفات لاستخدامها للتحقق من صحة بيانات تحقق مستقلة تماماً. وأظهرت النتائج إمكانية استخدام لغة النيلين للمواصفات لتوصيف أنواع البيانات المجردة وإمكانية استخدام أداة النيلين للتحقق من صحتها.

LIST OF PUBLICATIONS

[1] Ali, Nahid A., Amal A. Mirghani, and Abdelrasoul Y. Ibrahim. "Alneelain: A formal specification language." *Communication, Control, Computing and Electronics Engineering (ICCCCEE), 2017 International Conference on*. IEEE, 2017.

[2] Ibrahim, Abdelrasoul Yahya. "Specifying abstract data types a behavioral model, an axiomatic representation." *Computing, Electrical and Electronics Engineering (ICCEEE), 2013 International Conference on. IEEE, 2013.*

[3] Amal A. Mirghani, Nahid A. Ali, Abdelrasoul Y. Ibrahim, "Formal Specification Language." *International Journal of Engineering Sciences Paradigms and Research* ***(IJESPR)***, December 2017

[4] Abdelrasoul Y. Ibrahim Ali, Nahid A., and Amal A. Mirghani, "An Automated Technique to support specification generation and validation" International Journal of Science and Research, ISSN 2319-7064, June 2018

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ADTs	Abstract Data Types
AMN	Abstract machine Notation
BNF	Backus-Naur Form
COCOMO	Constructive Cost Model
ISO	International Organization for Standardizations
MIT	Massachusetts Institute of Technology
NSL	Alneelain Specification Language
NVT	Alneelain Validation Tool
SAT	Short for Satisfiability
SRS	Software Requirements Specification
ULS	Ultra Large Scale
UML	Unified Modelling Language

CHAPTER ONE

INTRODUCTION

1.1 INTRODUCTION

This thesis is a part of a team effort that involves three sub-objectives with me and other colleagues named Nahid Ahmed Ali and Amal Awad Mirghani. As my section in Specification generation and validation, while *Nahid* work in the specification and Program verification, whereas *Amal* work in Specification and program testing respectively. This introductory chapter gives a brief background of software engineering which gives brief concepts of software product, software life cycle, software specification which is a basis step toward develop software product, and the criteria that the software specification must satisfy. It gives problem state, Research objectives, research scope, research significance, research questions, research hypothesis, research methodology, and thesis organization.

1.2 SOFTWARE ENGINEERING BACKGROUND

Ever since it emerged in the late of nineteen sixties; the discipline of software engineering has set itself apart from other engineering disciplines, such as chemical engineering, mechanical engineering, civil engineering, electrical engineering [1], etc. however, software engineering has difference with others discipline in number of ways; including the pervasiveness of its product; the complexity of its products and process; the criticality of its applications; the intractability of its process lifecycle. As software engineering defined as in [1] is a systematic approach to the analysis design, assessment, implementation, test, maintenance, and re-engineering of software, that is, the application of engineering to software. Software lifecycle is the requirement specification, software design, programming, and testing [2]. Unlike the lifecycle of the project of the others engineering disciplines; software process lifecycle does not lend themselves to simple is highly iterative phases, hence, it has several models are defined and many methodologies for the definition and assessment of its phase; that is prone to backtrack, and suffer from poor visibility; software requirements specifications are an indispensable basis for any software sound analysis of functional properties of software [2]. The approach of identifying relevant stakeholders, eliciting relevant requirements, representing these requirements and combining them into a comprehensive specification are an important part of a software engineering [2]; software specifications remains the focus of much research for two of consequential reasons: First; because it is difficult to phase, that involves identifying requirements, classifying requirements, resolving conflicts, specifying requirements, validating the resulting specification for completeness, consistency, and minimality, etc [3]. Second, because any fault that arises in this phase later is likely to carry the high cost since it will impact all subsequent phases [2]. In addition, for the importance of carefully specifying software requirements prior to software product development, Specification is always played a central role in software engineering, not only because of its intrinsic technical interest but also because it is critical in determining the success or failure of software projects [3].

After software requirements analysis completed, software requirement specification is created, it describes in as much detail and an unambiguous a manner as possible, exactly what the product should do. The requirements specification document may well contain a model of the software functionality and behavior in form of specifications for the system [4], where should be appropriate, any performance, data considerations, any input/output detail. This specification should be in formal method as well. The specification must be valid, because the effects of invalid or valid specification can have far reaching effects on the software usefulness and cost, where the cost of maintaining or fixing invalid software specification can be a hundred of times greater than the original cost of getting it right.

Requirements specifications have remained an active research area for as long as the discipline of software engineering has been in existence [1]. Indeed, while new specification languages and specification methods are introduced, the demands placed on the software industry in terms of specifying ever more complex systems has also been getting heavier, thereby maintaining a wide technological gap that remains unfulfilled [2]. As software artifacts, software specifications must satisfy a number of criteria, including: First; Product Requirements: As products, the specification must satisfy the following criteria [2]:

Simplicity: The specification must be as simple as possible, to facilitate writing, reading, and analysis.

Formality: The specification must be formal, to fulfil their function in terms of verifying the correctness of candidate products.

Abstraction: The specification must be abstract, i.e. represent what properties candidate programs must satisfy, not how to satisfy these properties.

In other words, specifications must describe the WHAT (what a program must do) rather than the HOW (how to do it), the latter being a responsibility of the designer, not the specifier.

Second; Process Requirements: The process of specifying software requirements must satisfy two criteria, which are:

1-*Completeness:* The specification must reflect all the relevant stakeholder requirements.
2-*Minimality:* The specification must reflect nothing but the relevant stakeholder requirements.

1.3 PROBLEM STATEMENT AND SIGNIFICANT

For improvement in software-development efficiency, it is effective to describe software specifications well and to eliminate specifications errors before programming for following important reasons: Firstly, how to ensure that specifications are valid and reflect all the relevant requirements and constraints for completeness, consistency, and minimality. Secondly, because any fault that arises in this phase is likely to carry a high cost, due to the fact that an error in the requirements phase can have a negative impact on all subsequent phases.

1.4 RESEARCH OBJECTIVES

The main objective of this research is to design and develop a tool to help users generate and validate specifications written in algebraic notation. In [5] algebraic denotes sequence of events of a system which satisfy system properties such a system must do the following:

1. Designing a specification language around the trace notation.
2. Developing a compiler for that language and a user interface has an editor for specifications using the language.
3. Developing a tool that can validate a generated specification against captured validation information and use the tool to validate a wide range of ADTs.

1.5 RESEARCH SCOPE

This study is focus on specifying and validating abstract data types such as stack, queue, list, sequence, set, and multiset. So, we built specification language for specify abstract data types and built validation tool for validate them.

1.6 RESEARCH QUESTIONS

The most important questions of the research are: First, how do we represent our specification notation of the software to satisfy all the process and product requirements so that software engineers have no trouble adopting it and users have no trouble understanding it? Second how do we ensure that specifications are valid, and reflect all the relevant requirements and constraints?

1.7 RESEARCH HYPOTHESIS

We proposed that specifications are represented by axioms and rules where:

a) Axioms represent the behavior of the module for simple input histories
b) Rules relate the behavior of the module for complex input histories to its behavior for simpler input histories.

1.8 RESEARCH METHODOLOGY

The objective of this thesis is mainly targeted towards the development of an automated tool to support validation of ADTs that were specified using Alneelain specification language. This tool can handle and validate user data that selected in a random manner. As we want to achieve the research objective we should adopt a methodology that enables the researcher to obtain the research goals. The methodology described as follows:

1. Define and Specify ADTs in trace notation forms.
2. By using axiomatic specifications such ADTs that written in informal description resembles trace specification.
3. Generating Validation Data from the ADT's that should it comply with the specification being generated we argue that this data takes the form of (h, y) pairs, where h is an input history and y is the output that the validation team believes should be produced for the input history h.
4. Validating the specification against generated validation data and obtains validated axiomatic specification by checking them whether the generated specification does indeed comply with the independent validation data.

In particular, this done by the many steps and their consequences, which are the following:

1. Defining ADT's requirements to obtain ADTs requirements descript (ADTs Description).
2. Specifying ADT's using axiomatic specification (ADTs Specification).
3. Validating ADT's using independent validation data, while this validating data is produced by a different group who didn't participate in the previous step (ADTs validation).
4. Writing the proposed language of study (Alneelain Language).
5. Writing the specification of ADTs according to Alneelain Language.
6. Writing a part of compiler phase as a lexical analyzer to scans the source code as a stream of characters and converts it into meaningful lexemes inform of tokens.
7. The last step is building validation tool which has a user interface, It acts as the semantic analyzer, it takes the log file of ADT produced in the previous step which

includes all axioms and rules to checks whether independent user validation data is valid or not valid. Figure 1.1 shows the methodology.

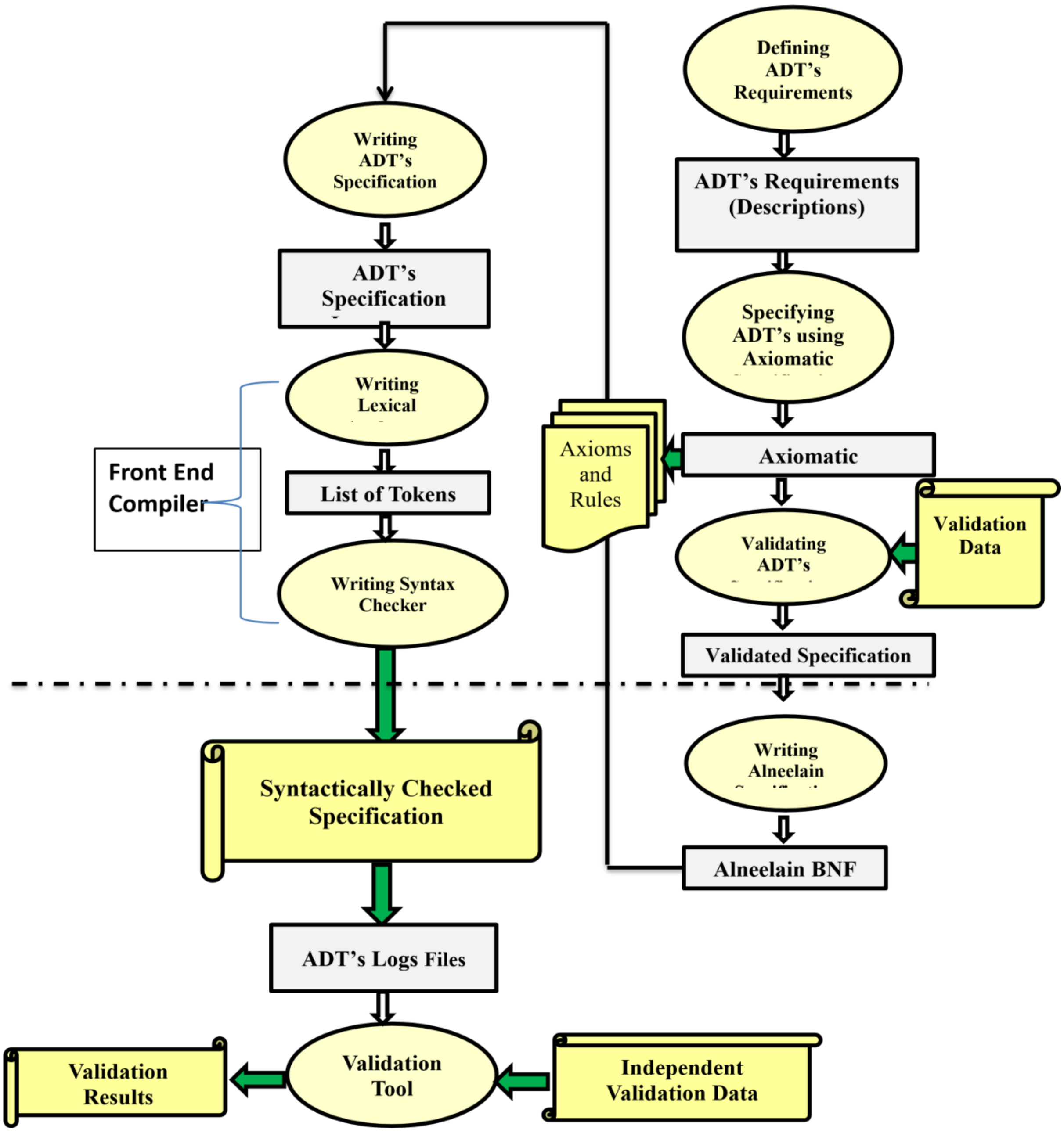


Figure 1.1: Depicts the above-mentioned steps and presents a clear roadmap of our work.

1.9 THESIS ORGANIZATION

This thesis is organized as follows: Chapter one gives an introductory that includes software engineering background which contains the concepts of software product, software life cycle, software specification, and the criteria that the software specification must satisfy. Also, this chapter gives problem state, Research objectives, research scope, research significance, research questions, research hypothesis, and research methodology. Chapter two gives the motivation of software engineering. Chapter three provides an overview of software specification and validation background, related work, model-based specification and Behavioral specification. Chapter four investigates a mathematical background which is includes relation and set and it explains the role of relation and set in specification and validation. Chapter five introduce A proposed specification language named Alneelain Specification Language to be used to specify abstract data types. Chapter six explores Alneelain compiler which is composes of two components of the language; they are lexical analyser and syntax checker. Chapter seven is about Alneelain Validation Tool NVT which can used to validate independent validation data. Chapter eight illustrates both NSL and NVT and explains their interfaces and how do they use by provides a user guide. Finally, chapter nine gives a conclusion of the research, limitations, and future work.

CHAPTER TWO
MOTIVATION OF SOFTWARE ENGINEERING

2.1 INTRODUCTION

This chapter explains the motivation and importance of software engineering as youngest discipline, and gives a brief history of them, the role of software engineering in industry and economical fields, it also explains software products complexity and expensive and difference between software products and other engineering products,

2.2 MOTIVATION

Software engineering is Systematic approach for developing software Methods and techniques to develop and maintain quality software to solve problems [6], also software engineering is the study of the principles and methodologies for developing and maintaining software systems [7], and software engineering is practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them [8]. However, software engineering differs from other engineering disciplines in varieties ways, which were explored below

2.2.1 Software engineering is modern discipline

Whereas civil engineering and mechanical engineering date back to antiquity, chemical engineering and electrical engineering date back to the eighteenth century, software engineering exhibit in the half of the twentieth century, hence is relatively young discipline, software engineering emerged in the second half of the twentieth century, hence is a relatively young discipline [2]. The concise history of this discipline can be divided into five broad eras, remaining approximately one decade each.

The first era is the sixties, the technology in pioneer. This era of technology marks the first time that researchers and practitioners came face to face with complexities, discrepancies, and malformations of software engineering. Software projects of this era were the venture into ventures into the uncharted ground, characterized by high levels of risk, unpredictable outcomes, and massive cost and schedule exceed.

The second era is the Seventies, the era of Structured Software Engineering [2]. This era is Structured Software Engineering. This era is described by the general belief that software engineering problems are of a technical and that if we and that if we improve techniques for software specification, design, testing and verification, all software engineering problems would be resolved. Given that structure is our main intellectual tool for dealing, this era has seen the emergence of a wide range of structured techniques, including structured programming, structured design, structured analysis, structured specifications, and so on. Such programming languages that were dominant in this era are Pascal and C.

The third era is the Eighties, Knowledge-based software engineering this era is described by the realization that software engineering problems are of managerial and organization nature more than technical nature. This realization was synchronous with the emergence of Fifth generation Computing initiative, which started in Japan and disseminates across the world (the USA, Europe Canada), and was focused on thinking machines designed with the inclusive use of artificial intelligence techniques. This general approach penetrates the discipline of software engineering techniques, with the emergence of knowledge-based software engineering techniques. Some of the programming languages that were prevailing in this period are PROLOG, LISP, and ADA. The Nineties, Reused based software engineering [2]. As it becomes increasingly clear that fifth generation computing was not delivering on its promise.

Forth, the nineties reuse based Software Engineering. As it became more and more clear that fifth generation computing was not delivering on its time, and worldwide fifth generation initiatives were fading, software researchers and practitioners turned their attention to reuse as the possible rescuer of the discipline. Software engineering is, anyway, the only discipline where reuse is not an integral part of the routine engineering process [2]. It was felt 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 up growth was concurrent with the emergence of object-oriented programming, which supports a bottom design discipline that facilitates product reuse. The programming languages that were dominant in this era are C, C++, and Smalltalk.

The first decade of the Millennium is Lightweight software engineering, while software reuse is not practical as the general paradigm in software engineering; it is feasible in limited application domains, giving rise to product line engineering. Other attributes of this era include Java programming, with its focus on web applications, agile programming, with its focus on rapid and flexible response to change, and component-based software engineering, with its focus on software architecture and software composition. The programming languages that were dominant in this era are Java, C++, and Python. Perhaps as result of this young and eventful history, the discipline of software engineering is characterized by a number of paradoxes and computer-intuitive properties. In this chapter, we will explore software characterized as software motivation [2].

2.2.2 An industry under stress

Nowadays, the software runs all aspects of our life and accounts for the large and increasing share of the world economy [2]. This trend started slowly with the advent of computing in the middle of the twentieth century and was further precipitated by the emergence of the World Wide Web at the end of the twentieth and beginning of the twenty-first century. This phenomenon has spawned a great demand for software products and services and generated a market pressure that the software industry takes pains to cater to many fields of science and engineering (such as bioinformatics, medical informatics weather, forecasting, modelling, and simulation, etc.) are also dependent on software that they can almost be considered as mere applications of software engineering. Also, it is possible to observe that many computer science curricula are slowly inching towards more software engineering contents at the expense of traditional theoretical material that may be perceived as fewer trends by starting software engineering degrees in computer science departments or by starting complete software engineering departments alongside traditional computer science departments [9].

Concurrent with a widening demand for software to serve ever broader needs, we are also witnessing higher and higher expectations in terms of products quality. As software takes on ever more vital functions in life-critical and mission-critical applications, and in applications that carry massive financial stakes, it becomes increasingly important to ensure that software products fulfil their function with a high degree of dependability. This requires that we deploy a wide range of technique, including, first, process controls to ensure that software products are developed and evolved according to certified mature processes, second, products controls to ensure that software products meet quality standards commensurate with their application domain requirements; this is achieved by a combination of techniques, including static analysis, dynamic testing reliability estimation, fault tolerance, etc. In summary, it is fair to argue that the software industry is under massive stress to deliver both quantity and quality; as we in the following subsequence sections this both difficult and expensive.

2.2.3 Large complex products

The demand for complex hardware and software systems has increased rapidly than the ability to design, implement, test and maintain them [10] as Michael said. Not only is it critical for us to build software products that are of high quality, it is also very difficult, due to their size and complexity. Completing projects of this kind of size is not only a major engineering undertaking, it is also a major organizational challenge; it is estimated that the production of the Windows Server 2003 involved two thousand software personnel (programmers, analysts, engineers) for development and two thousand four hundred software personnel for software testing.

A panel convened by the Software Engineering Institute (www.sei.cmu.edu) in 2005-2006 to analyse software systems of the future and draw a research agenda to manage such systems estimates that future software systems are expected to have sized up to a billion lines of code. Along with this dry measure of size, such systems will be large in terms of other dimensions, such as (www.sei.cmu.edu/uls/): the amount of data stored, accessed, manipulated and refined; the number of connections and interdependencies; the number of hardware elements; the number of computational elements; the number of system purposes and user perception of these purposes; the number of routine processes, interactions, and emergent behaviours; the number of overlapping policy domains and enforceable mechanisms; and the number of parties involved in the operation of the system (developers, maintainers, end users, stakeholders, etc.).

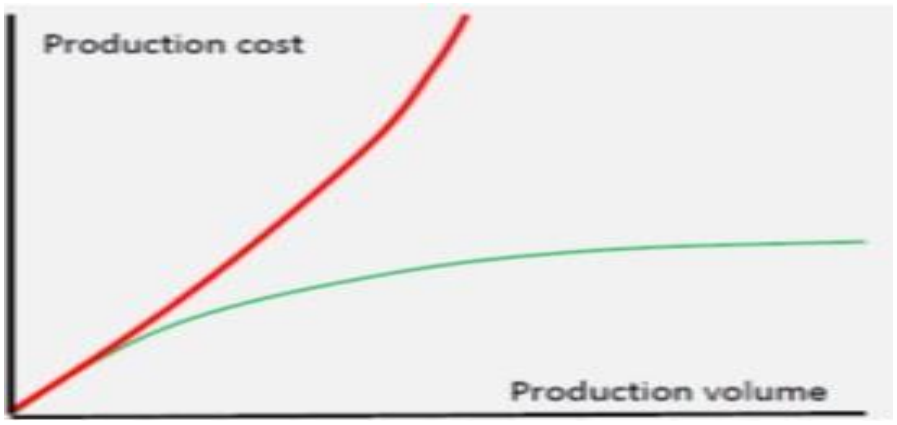
Size changes everything: such systems (referred to as Ultra Large Scale systems, or ULS) challenge all our knowledge and assumptions about software, and are estimated to have a number of distinguishing features, such as: *Decentralization* in fundamental dimensions such as decentralized development, decentralized evolution, decentralized operation, etc. *Conflicting*, unknown and diverse, whereas the traditional view in software engineering is that requirements must be analysed, compiled and specified prior to software design and development, the view taken by the ULS approach is that at no time can we claim that all relevant requirements have been collected and specified. *Continuous* evolution and deployment. Whereas the traditional view of software engineering is that a software product proceeds sequentially through successive phases of development, then maintenance, then phase out, ULS systems are developed, evolved and deployed concurrently (made up of parts that are at different stages in their evolutionary process). *Heterogeneous*, inconsistent, changing elements. Whereas a traditional software product is developed as a cohesive monolithic system by a development team, ULS systems emerge as the aggregate of many components, which may have evolved independently, using different paradigms, different technologies, by different teams, from different stakeholder classes. In addition, different components of the system are expected to evolve relatively independently. *Deep erosion* of the people-system boundary. Whereas traditional systems are defined in terms of a distinct boundary that separates them from the outside world, ULS systems are envisioned to include human users as an integral part. So that when a user interacts with a ULS system, he/she may be engaging human actors along with system behavior *Failure* is normal and frequent. Whereas in traditional software systems we think of failures as exceptional events and consider that failures avoidance is contingent upon fault removal, ULS systems, we take a broader view of successful (failures free) operation, which does not exclude the presence of faults, but makes provision for system redundancy and requirement no determinacy to make up for presence of faults.

2.2.4 Expensive product

Not only are software products very large and complex, they are very expensive to produce. Of course, if a product is large, one expects it to be costly, but what is surprising is that the unitary cost of software, i.e. the cost per line of code, is itself very high [9].

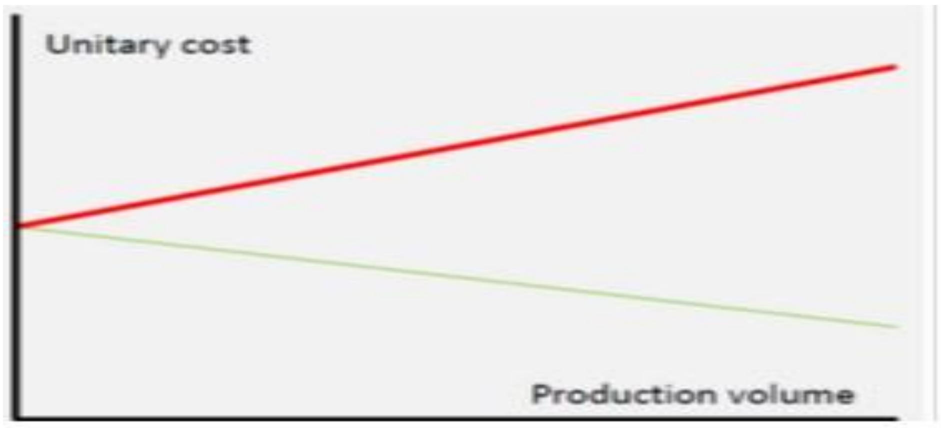
Whereas any programmer you ask may tell you that they can produce a hundred lines of code in a day or a week, a more realistic figure, across all areas of software development, is closer to about 100 lines of code per day, or about 300 lines of code per person-month. This figure includes all costs that are spent producing software, including the cost of all phases of the software lifecycle, from requirements analysis and specification to software testing [9]. If we assume the cost of a person-month to be twenty thousand dollars (in salary, fringe benefits, and related expenses), this amounts to about 100 $ per line of code. If for the sake of argument, we apply Boehm's COCOMO cost estimation model to a bespoke software project of size 500000 source lines of code developed in embedded mode (the hardest/most costly development mode), we find 80 source lines of code per person-month.

In most other engineering disciplines, one way to mitigate costs is to use economies of scale, i.e. to produce in such a large volume as to lower the unitary cost, Economies of scale are possible because in the engineering disciplines, the production increases. The same process applies in software engineering: If we invest resources to acquire software tools, to train software professionals, or to set up a programming environment, then the more software we produce the better our investment is amortized. But in software we are also dealing with a phenomenon of diseconomy of scale: the more software we produce within a single product, the more interdependencies we create between the components of the product, so that the unitary cost (per line of code) of large software products is larger than that of smaller products. This phenomenon of diseconomy of scale overrides the traditional economy of scale [9]; the net result is a diseconomy of scale, which is all the acuter that the software product is larger or more complex; Figure 2.1 and Figure 2.2 explain diseconomies of scale in software engineering [2].



🟥 Software engineering 🟩 otherengineering disciplines

Figure 2.1: Production cost as a function of production Size.



🟥 Software engineering 🟩 another engineering discipline

Figure 2.2: Unitary cost as a function of production Size [2]

Many of these costs are mitigated nowadays by the use of a variety of coarsely grained software development methods, which proceed to build software by composing existing components, rather than by painstakingly writing code from scratch line by line. Another trend that is emerging recently to address software cost and quality is the use of so-called agile methodologies. These methodologies control the costs and risks of traditional lifecycles by following an iterative, incremental, flexible lifecycle, where the user participates actively in the specification and development of successive versions of the targeted software product [9].

2.2.5 Absence of reuse practice

In the absence of economies of scale, one would hope to control costs by a routine discipline of reuse; in the case of software, it turns out that reuse is also very difficult to achieve on a routine basis. In any engineering discipline, reuse is made possible by the existence of a standard product architecture that is shared between the producer and the consumer of reusable assets: for example, automobiles have had a basic architecture that has not changed for over a century; all cars have a chassis, 4 wheels, an engine, a battery, a transmission, a cab, a steering column, a braking system, a horn, an exhaust system, shock absorbers, etc. Thanks to this architecture, the design of a new car is relatively simple, and is driven primarily by design and marketing considerations; the designer of a new model does not have to reinvent a car from scratch, and can depend on a broad market of companies that provide standard components, such as batteries, tires, spare parts, etc. The standard architecture of a car dictates the market structure and creates great efficiencies in the production and maintenance of a car [9].

Unfortunately, no standard architecture exists in software products; this explains, to a large extent, why the expectations that software engineering researchers and practitioners pinned on a discipline of software reuse never fully materialized. large software reuse initiatives were launched in the last decade of the last century, making available a wide range of software products, and sophisticated search and assessment algorithms; but they were unsuccessful, because software reuse requires not only functional matching between the available components and the requirements of the user but also architectural matching, which was often lacking. The absence of a standard architecture of software products also explains why software product lines have achieved some degree of success: product line engineering is a form of software reuse that is practiced in the context of a narrow application domain, in which it is possible to define and enforce reference architecture. As an example, if we define a product line of e-commerce systems, we may want to define the reference architecture as being composed of the following components: a web front-end; a shopping cart component; an order processing component; a banking component;

a marketing and recommendations component; a network interface; and a database interface [9].

2.2.6 Fault prone designs

The design space of software products is broader than that of any other engineering product. As a result, it is much more difficult to codify and standardize best practices in software engineering than it is in any other engineering discipline.

2.2.7 Paradoxical economics

While technology can change quickly, getting your people to change takes a great deal longer. That is why the people-intensive job of developing software has had essentially the same problems for over 40 years [11].

2.2.8 A Labor-intensive industry

If we consider the cost of an automobile, for example, and ponder the question of what percentage of this cost is due to the design process and what percentage is due to manufacturing, we find that most of the cost (more than 99%, perhaps) is due to manufacturing. Typically, by the time one buys a car, the effort that went into designing the new model has long since been amortized by the number of cars sold; what one is paying for is all the raw materials and the processing that went into manufacturing the car [9]. By contrast, when one is buying a software product, one is paying essentially for design effort, as there are no manufacturing costs to speak of loading compact disks or downloading program files [9]. The following table 2.1 shows, summarily, how the cost of a software product differs from the cost of another engineering product in terms of distribution between design and manufacturing:

Table 2.1: Lifecycle Cost Distribution: Design vs. Manufacturing.

	Software engineering	Other engineering
Design	> **99**%	< 1%
Manufacturing	< 1%	> **99**%

2.2.9 Absence of automation

The Labor intensive nature of software engineering has an immediate impact on the potential to automate software engineering processes. In all engineering processes, one can achieve saving in manufacturing by automating the manufacturing process or at least streamlining it, as in assembly lines. This is possible because manufacturing follows a simple, systematic process that requires little or no creativity. By contrast, design cannot be automated because it requires creativity, artistic appreciation, aesthetic sense, etc.

Automating the manufacturing process has an impact in traditional engineering disciplines because it helps reduce a cost factor that accounts for more than 99% of production costs, but it has no impact in software engineering because it affects less than one percent of production costs. Hence, the automated development of software products is virtually impossible in general.

The only exception to this general rule is the development of applications within a limited application domain, where many of the design decisions may be taken a priori when the automated tool is developed and hardwired into the operation of the tool. One of the most successful areas of automated software development is compiler construction, where it is possible to produce compilers automatically, from a syntactic definition of the source language, and relevant semantic definitions of its statements [9].

2.2.10 Limited quality control

The lack of automation, hence the absence of process control, make it difficult to control product quality. Whereas in traditional engineering disciplines the production process is a systematic, repeatable process, one can control quality analytically by certifying the process, or empirically by statistical observation.

2.2.11 Unbalance lifecycle cost

In the most engineering disciplines, products are produced in large volume and are generally assumed to behave as expected; in software engineering, due to the foregoing discussion, such an assumption is unfounded, and the only way to ensure the quality of software product is to subject that product to extensive analysis. This turns out to be an expensive proposition, in practice, and the source of another massive paradox in software engineering economics. Whereas testing (and more generally, verification and quality assurance) takes up a small percentage of the production cost of an engineering artifact, it accounts for a large percentage of the lifecycle cost of a software product [9]. Table 2.2 shows a comparison between software engineering and other engineering in term of development and testing.

Table 2.2: Lifecycle distribution: development vs. Testing.

	Software engineering	Other engineering
Development	≈ 50%	$> \mathbf{99}\%$
Testing	≈ 50%	$< \mathbf{1}\%$

Good software engineering practice dictates that more effort ought to be spent on upfront specification and design activities and that such upfront investment enhances product quality and lessens the need for massive investment in a posteriori testing. While these practices appear to be promising, they have not been used sufficiently widely to make a tangible impact; so that software testing remains a major cost factor in software lifecycles.

2.2.12 Unbalanced maintenance cost

It is common to distinguish, in software maintenance between several types of maintenance activity; the two most important types (in terms of cost) are*: Corrective maintenance*, which aims to remove software faults. *Adaptive maintenance*, which aims to adapt the software product to evolving requirements.

Empirical studies show that adaptive maintenance accounts for the vast majority of maintenance costs. This contrasts with other engineering disciplines, where there is virtually no adaptive maintenance to speak of: it is not possible for a car buyer to return to the dealership to make her/his car more powerful, add seats to it, or make it more fuel efficient. Hence it is possible to distinguish between software products and other engineering products by the distribution of maintenance, as shown in the following table 2.3.

Table 2.3: Maintenance Cost Distribution: Corrective vs. Adaptive.

	Software engineering	Other engineering
Correction	≈ 20%	> 99%
Adaptive	≈ 80%	< **1**%

2.3 SUMMARY

This chapter presents software engineering discipline and its important role that it plays contrast with other traditional engineering disciplines. The chapter explains many of software engineering characteristics such as limited of quality control, absence of reuse unbalanced of its lifecycle cost, etc. that make it varies from other engineering disciplines. Also, it presents the increasing demand for software engineering products, making software engineering of great importance in engineering fields.

CHAPTER THREE
SOFTWARE SPECIFICATION GENERATION AND VALIDATION

3.1 INTRODUCTION

In this chapter we review the body of the work. Next will review the existing of two approaches for specification of abstract data type; Model-based specification and behavioral specification, B-Trees [12] specifications are one of the most used ADT in system development in which the specification described in three form: system architecture, static behavior, and dynamic behavior, the later one is our focus which describe the operations of ADTs, whereas system architecture of B-Trees lists all component needed to describe the behavior before specifies them in details.

3.2 RELATED WORK

3.2.1 Specification concepts

The specification of a software product is a description of the characteristic's product must to have fulfilled its purpose. The specification is usually derived by identifying all the relevant stakeholders of the planned software product, eliciting the all requirements that they expect the product to meet, formulating and combining these requirements, and compiling them into a coherent document. These make specification is critical phase of software lifecycle [13]. While specifications typically pertain to functional and operational requirements, we focus primarily on functional requirements rather than operational requirements, that is to pertain the input and output behavior of software product. As abstractions are intangible, the specification gives us an idea that an abstraction is independent of any of its implementations [14].

Also we can define that specification is a description of the system be developed it abstract away from how a behavior is obtained and just show declaratively what is being expected [15].The specification of any a produce software is the description of properties that the software product must have meets its purpose. The specification is commonly derived by identifying all the relevant customers of the existing software product, eliciting the requirements that they expect the product to meet, formulating and combining these requirements. As a product, specification must meet two conditions [9]: the first one is *Formality;* the specification must be represented in such a way as to describe precisely what functional behavior is required. And second is *Abstraction;* the specification must describe what requirements the software product must satisfy, not how to satisfy them, in other words, it must focus on *what* versus. *How*, the second one is being the prerogative of the designer. Making software starts from requirements analysis, i.e., eliciting and specifying the goals of the stake-holders and learning about the domain in which and about which the software is going to be developed [9]. The requirements have to be put down in terms of models or specifications [2]. An SRS (a software requirement specification) should be unambiguous, consistent, and verifiable. An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation [2]. Requirements are often written in natural language. Natural language is inherently ambiguous. One way to avoid the ambiguity inherent in natural language is to write the SRS in a particular requirements specification language. By specification language we understand a formal system of syntax and semantic of that language [16].

An SRS is internally consistent if, and only if, no subset of individual requirements stated therein conflict [2].

3.2.2 Specification method

A specification method is a sequence of activities leading to the development of a Product, called a specification. A method should provide enough guidance on how to conduct the activities and on how to evaluate the quality of the final product. A specification is a precise description, written in some notation (language), of the client's requirements. A notation is said to be formal if it has a formal syntax and formal semantics. A notation is said to be semi-formal if it only has a formal syntax. Several characteristics of a system can be specified. One may distinguish between functional requirements, efficiency requirements and implementation requirements. Functional requirements address the input-output behavior of a system. Efficiency requirements address the execution time of a system. The client may be interested in specifying a time bound for obtaining a response from the system. Some authors argue that [17] a specification without time bounds is not an effective specification: indeed, strictly speaking, if the specification does not include a time bound, the implementation may take an arbitrary duration to provide a response. It is impossible to distinguish between an infinite loop and a program that takes an arbitrary time to respond. Implementation requirements address issues like the programming language to use, the software components to reuse, the targeted hardware platform, the operating systems. The methods described in this book address functional requirements

3.2.3 Formal specification

The term formal method is used to refer to activities that rely on mathematical representation of software including formal system specification, specification analysis and proof as well as validation and program verification. All of these activities are dependent on formal specification of the software. A formal software specification should explained in language whose vocabulary, syntax and semantics are formally defined [18]. These are need for mathematical concepts whose properties are well understood. The branches of mathematics used in discrete mathematical concepts are drawn from set theory, logic and algebra. In the 1980s, many software engineering researchers proposed that using formal development methods was the best way to improve software quality [19]. They argued that the detailed analysis and validation are essential part of formal methods would lead to programs with fewer errors and which were more suited to users' needs [20]. They predicted that, by the 21st century, a large proportion of software would be developed using formal methods [19]. However, this prediction has not come true for the four following reasons are:

1. formally defining types of data, and mathematical operations on those data types
2. abstracting implementation details, such as the size of representations (in memory) and the efficiency of obtaining outcome of computations
3. formalizing the computations and operations on data types
4. Allowing for automation by formally restricting operations to this limited set of behavior and data types.

In This section, it is possible to review the two major approaches of specification: *algebraic specification* and *mode-based specifications* [21]. which they are explaining as follows:

- Model based specifications, which represent specifications by describing an abstract model of the behavior of software systems,
- Algebraic specifications, which represent specifications by describing the external observable behavior of software systems.

3.2.4 Model-based specification

Model-based specification is an approach to formal specification where the system specification is expressed as a system state model. This state model is constructed using well-understood mathematical entities such as sets and functions. System operations are specified by defining how they affect the state of the system model. The most widely used notations for developing model-based specifications are [22] [23] [24]. These notations are based on typed set theory. Systems are therefore modelled using sets and relations between sets. On the other hand, there is a good definition for Model based specification which says: A Model based specification is a form of specification, usually software specification, which is developed by creating a mathematical model of that system. Typically the mathematical model is expressed in terms of objects and operations, and these are defined using such mathematical concepts as sets, relations, and functions [25] That mean a model of the system is constructed using well-understood mathematical entities such as sets and sequences, the state of the system is not hidden (unlike algebraic specification), and state changes are straightforward defined. In algebraic approach was originally designed for the definition of abstract data type interfaces. In an abstract data type, the type is defined by specifying the type operations rather than the type representation. Therefore, it is similar to an object class. The algebraic method of formal specification defines the abstract data type in terms of the relationships between the type operations. Gutting [14] first discussed this approach in the specification of abstract data types. Cohen [26] shows how the technique can be extended to complete system specification using an example of a document retrieval system. The body of the specification has four components.

1. An introduction that declares the sort (the type name) of the entity being specified. A sort is the name of a set of objects with common characteristics. It is similar to a type in a programming language. The introduction may also include an 'imports'

declaration, where the names of specifications defining other sorts are declared. Importing a specification makes these sorts available for use.

2. A description part, where the operations are described informally. This makes the formal specification easier to understand. The formal specification complements this description by providing an unambiguous syntax and semantics for the type operations.
3. The signature part defines the syntax of the interface to the object class or abstract data type. The names of the operations that are defined, the number and sorts of their parameters, and the sort of operation results are described in the signature.
4. The axioms part defines the semantics of the operations by defining a set of axioms that characterize the behavior of the abstract data type. These axioms relate the operations used to construct entities of the defined sort with operations used to inspect its values.

The process of developing a formal specification of a sub-system interface includes the following activities:

1. Specification structuring Organize the informal interface specification into a set of abstract data types or object classes. You should informally define the operations associated with each class.
2. Specification naming Establish a name for each abstract type specification, decide whether or not they require generic parameters and decide on names for the sorts identified.
3. Operation selection Choose a set of operations for each specification based on the identified interface functionality. You should include operations to create instances of the sort, to modify the value of instances and to inspect the instance values. You may have to add functions to those initially identified in the informal interface definition.
4. Informal operation specification Write an informal specification of each operation. You should describe how the operations affect the defined sort.
5. Syntax definition Define the syntax of the operations and the parameters to each operation. This is the signature part of the formal specification. You should update the informal specification at this stage if necessary.

6. Axiom definition Define the semantics of the operations by describing what conditions are always true for different operation combinations.

3.2.5 Algebraic specification

The simple algebraic techniques described in the previous section can be used to describe interfaces where the object operations are independent of the object state. That is, the results of applying an operation should not depend on the results of previous operations. Where this condition does not hold, algebraic techniques can become cumbersome. Furthermore, as they increase in size, A researcher found that algebraic descriptions of system behavior become increasingly difficult to understand. Algebraic specification (often known as behavioural specification) [27] is a software engineering technique for formally specifying system behavior (i.e. the system is described in terms of operations and their relationships). Algebraic specification seeks to systematically develop more efficient programs by an algebraic specification achieves these goals by defining one or more data types, and specifying a collection of functions that operate on those data types. These functions can be divided into two classes:

1. Constructor functions: functions that create or initialize the data elements, or construct complex elements from simpler ones
2. Additional functions: functions that operate on the data types, and are defined in terms of the constructor functions.

3.3 SPECIFICATION LANGUAGES

There are many specification languages which are present in the following sections

3.3.1 Z specification language

The Z (pronounced *Zed)* language is a formal specification language that makes it easier to write mathematical description of complex dynamic system such as software. The descriptions are usually smaller and simpler than any programming language can provide. They should contain a mixture of formal and informal parts. Z was developed in Paris, France and Oxford, England. Z is a specification language for structuring mathematical specification of computer systems. It has rigorous settheoretic foundations and has been used successfully in specifying computer systems, including business information systems. The Z notation has been standardized by ISO (International Organization for Standardization) in 2002 under standard ISO/IEC 13568:2002, entitled Information Technology – Z Formal Specification Notation - Syntax, Type System and Semantics [28].

A specification in Z is meant to be descriptive, i.e., to specify what the state of the system and its behavior are and not how they are to be implemented. It provides a very powerful specification language that allows for very abstract and non-executable specifications. A specification in Z is model-based, i.e., it focuses on the specifying the state of the system and possible transformations thereon [28]. However, Z can't be used to specify such a Behavioral specification. In contrast our language could be used as Behavioral specification language which is describing the external observable behavior of software systems; this made two approaches complement each other.

Z is model-oriented approach which a specific model of the state of abstract machine. Its operations are defined in term of that state [29]. The main properties in Z language is that have included a mathematical notation that used for formal specification. To structure Z specification and describes state transitions and states, the schema calculus was used [29]. It is visually striking, and consists essentially of boxes, with these boxes or schemas used to describe operations and states. The schema calculus enables schemas to be used as building blocks and combined with other schemas. In Z the identifiers may be composed of upper- and lower-case letters, digits, and the underscore character; must begin with a letter.

▸ Identifiers may have suffixes:

- **?** Means an input variable
- **!** Means an output variable
- **′** Means a new value (i.e., the after-operation value) ▸ Schema identifiers may have prefixes:
- Δ Means the state has changed (described later)
- Ξ Means no change in the state (described later)

Figure 3.1 shows the specification of the positive square root a real number.

```
–SqRoot————
 num?, root! : ℝ
———
num? ≥ 0
root!² = num?
root! ≥ 0
————————
```

Figure 3.1: Specification of square root using Z language.

The figure shown that an input real number is num?, and output number of square root is root!; both are real numbers. In predicate the input and output number must be greater than zero and root!^{2} is equal to num?.

As we saw in Z a specification is done by using schemas which provide structure to specifications. The schema consists of two parts

1. A declaration of variables
2. A predicate constraining the value of the variables

The schema can be displayed in two types as in following both figure 3.2 and 3.3:

$$Schema \mathrel{\widehat{=}} [\, declarations \mid predicates \,]$$

Figure 3.2: Z specification schema type1.

```
┌─Schema────────────────────────
│ declarations
├───────────────
│ predicates
└───────────────────────────────
```

Figure 3.3: Z specification schema type2.

A Z specification typically defines a number of state and operation schemas. A state schemas groups together variables and define the relationship that holds between their values. At any instant, these variables define the state of that part of the system which they model. An operation schema defines the relationship between the ‘before’ and ‘after’ states corresponding to one or more schemas. Inferring the operation schemas that may affect a particular state schema requires examining the signature of all operation schemas. However, this is impracticable in large specifications. The major feature of object –Z is the class which encapsulates a state schema with the operations which may affect that state. Instances of a class are called object syntactically; a class is named box with the constituents of the class are described and related.

The state schema is nameless and contains declarations (the state variables) and predicate (the state invariant). The state variables and constants are collectively called the attributes or features. The attributes and predicate of the state schema are implicitly included in the declaration and predicate part, respectively. Consider the following object-Z of class queue specification. Figure 3.3 shows a generic queue state that modeled by state schema *Queue[Item]* is the types of the items in the queue. Figure 3.4 shows the initialization of the queue, the queue is empty and no items have been joined to the queue this modeled by schema *QueueInit[Item]*. Figure 3.6 shows the join operation which modeled by schema *join[Item]*. Figure 3.7 shows the leave operation which modeled by *leave[Item]*. Final, figure 3.8 shows the interface of the queue which comprises the state variables *count*, the initial state schema and operations *join* and *leave*.

```
┌─ Queue[Item] ──────────────────────────────
│ items : seq Item
│ count : ℕ
└────────────────────────────────────────────
```

Figure 3.4: state schema of the queue

```
┌─ QueueInit[Item] ──────────────────────────
│ Queue[Item]
├──────────────
│ items = ⟨ ⟩
│ count = 0
└────────────────────────────────────────────
```

Figure 3.5: initialize the queue.

$Join[Item]$

$\Delta Queue[Item]$

$item? : Item$

$items' = items \frown \langle item? \rangle$

$count' = count + 1$

Figure 3.6: join operation in queue.

$Leave[Item]$

$\Delta Queue[Item]$

$item! : Item$

$items = \langle item! \rangle \frown items'$

$count' = count$

Figure 3.7: leave operation in queue.

$Queue[Item]$

$\upharpoonright (count, \textsc{Init}, Join, Leave)$

$items : \mathrm{seq}\ Item$

$count : \mathbb{N}$

$\textsc{Init}$

$items = \langle \rangle$

$count = 0$

$Join$

$\Delta(items, count)$

$item? : Item$

$items' = items \frown \langle item? \rangle$

$count' = count + 1$

$Leave$

$\Delta(items)$

$item! : Item$

$items = \langle item! \rangle \frown items'$

Figure 3.8: interface of queue comprises initial state, join, and leave.

3.3.2 Alloy specification language

Alloy is a structural modeling language that has been developed by Daniel Jackson's group at MIT. Alloy is based on first-order logic to express complex structural constraints and behavior. It is designed to precisely describe the specification and the modeling of the system [30].

Alloy was created by making use of the core package of the UML meta-model and the well-formed constraints of UML. The subset selected by Alloy from UML is consistent. Alloy is a little language with only a few modeling notations, which is easy to read and write. The grammar and the syntax of Alloy are also easy to grasp. . Alloy has formal syntax and semantics. Alloy has a constraint analyser which can be used to automatically analyser properties of Alloy models. An Alloy model consists of a group of sets and relations, defined in a first-order relational logic, with additional constructs so you can name and reuse sets and relations, make assertions, and tell the Alloy analyzer how you want everything checked The Alloy analyser will then generate interesting instances of your model and check them, using a sophisticated approach based on SAT solvers. The analyser won't be able to generate all instances of your model (unless it is a very simple model indeed), since in general there are an infinite number of them. So it will generate the ones that are likely to be most interesting, within the size limits you specify [31].

The following figure 3.9 shows an Alloy object model for a family tree

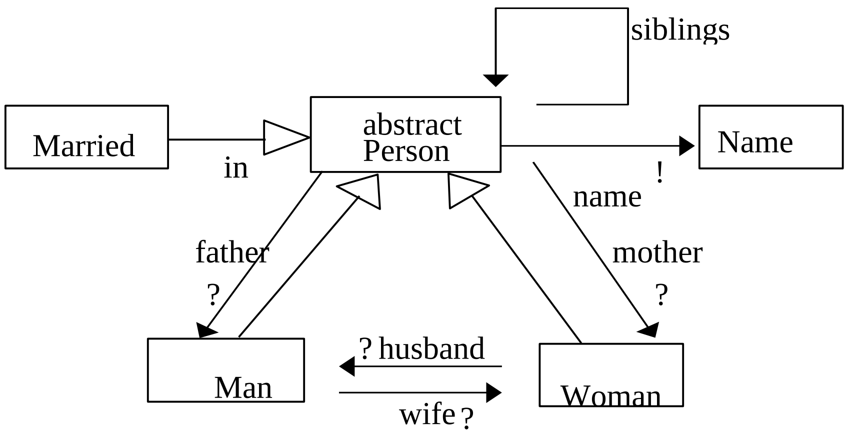


Figure 3.9: Family tree in alloy language.

Each box denotes a set of objects (atoms), which are corresponding to the object class, these are called signature, and object is an abstract and an unchanging entity. The state of the model is determined by the relationships among objects and the membership of objects in sets. This model can be changed in time; an arrow with unfilled head denotes subset. Man, Woman, Married are subsets of Person.

The key word **extends** indicates disjoint subsets. This is the default, if a subset is not labeled it is assumed to extend Man and Woman are disjoint sets (their intersection is empty). There is no Person who is a Woman and a Man. The keyword **in** indicates subsets, not necessarily disjoint from each other (or other subsets that extend).

- Married and Man are not disjoint
- Married and Woman are not disjoint

In Alloy sets of atoms such as Man, Woman, Married, Person are called **signatures.** A signature that is not subset of another signature is a top-level signature. Top-level signatures are implicitly disjoint; Person and Name are top-level signatures. They represent disjoint sets of objects. Extensions of a signature are also disjoint

– Man and Woman are disjoint sets

An abstract signature has no elements except those belonging to its extensions; hence there is no Person who is not a Man or A woman. Arrow with a small filled arrow head is denoting relations, i.e. name is a relation that maps Person to Name.

Markings at the ends of relation arrows denote the multiplicity constraints

– * means zero or more (default)
– ? means zero or one – ! means exactly one
– + means one or more
– If there is no marking, the multiplicity is *

Name maps each Person to exactly one Name, Name maps zero or more members of Person to each Name.

Alloy is actually a textual language. The graphical notation is just a useful way of visualizing the specifications but it is not necessary for the specification. In Alloy, the textual representation represents the model completely, i.e., the graphical representation is redundant. Following module shows the model in alloy language.

```
module language/Family

            sig Name { }
abstract sig Person {

             name: one Name,

 siblings: Person,

             father: lone Man,

 mother: lone Woman

             }

sig Man extends Person {

             wife: lone Woman

 }

sig Woman extends Person {


 husband: lone Man

 }

            sig Married in Person {

 }
```

Textual representation starts with **sig** declarations defining the signatures (sets of atoms). You can think of signatures as object classes, each signature represents a set of objects. The multiplicity is explained as follows:

– **set**	zero or more
– **one**	exactly one
– **lone**	zero or one
– **some**	one or more

The fields define relations among the signatures; similar to a field in an object class that establishes a relation between objects of two classes. Visual representation of a field is an arrow with a small filled arrow head. After the signatures and their fields, facts are used to express constraints that are assumed to always hold; fact are constraints that restrict the model and any configuration that is an instance of the specification has to satisfy all the facts. Following module shows family tree model restricted with facts.

```
            module    language/Family
sig    Name { } abstract sig   Person {


 name:    one Name,   siblings:   Person,

             father:   lone Man,

 mother:  lone Woman

             }
sig Man extends Person {

             wife: lone Woman

 }

            sig Woman extends Person {

 husband: lone Man
```

```
            } sig Married extends Person {
  }
fact {
  no p: Person | p in p.^(mother + father)
                     wife = ~husband
}
```

3.3.3 B specification language

The B method is method of software development based on B, a tool supported formal method based around an abstract machine notation [32], used in the development of computer software. It was originally developed by Jean-Raymond Abrial . B is related to the Z notation (also originated by Abrial) and supports development of programming language code from specifications. The B method has an associated specification notation, the so-called Abstract Machine Notation (AMN) [33]. This specification notation is classified as a state-based notation and is quite similar to such well-known formal notations as Z. The similarities between Z and B arise from the fact that the creator of the B method Jean-Raymond Abrial is also the author of Z. Compared to the specification notation of Z, AMN is more appealing to programmers, as it includes such statements as "IF THEN ELSE " and "WHILE " along with non-deterministic specification statements such as nondeterministic choice "ANY " [33] . The B method has three development stages: the specification, the refinement, and the implementation. In contrast the specification between B and Alloy we found that specification of B notation is a top down approach supported by its tool called theorem prover tool whereas the specification of alloy notation is state-based supported by its tool called Alloy Constraint Analyser [34].

In contrast all these specification languages discuss above to our approach, the reader will found that all these language are model- based specification rather than behavioral specification which can satisfied by our languages so as to complete two approaches each other.

3.3.4 Parnas trace specification

Trace specifications are conducive to good programming practice by requiring program output to be specified solely in terms of input, the trace method not only forces designers to make any information shared by two or more modules part of an explicit interface [35], it also discourages unnecessary modular coupling by focusing the designer's attention on such shared information [36]. This makes independent implementation of modules possible and leads to understandable software that is easier to maintain [36]. However, the trace specification it contains neither artifact from a particular language for presenting algorithms nor artifacts from a particular algorithm.

In the trace specification method, programs are specified by describing three properties they possess:

1) What do the access procedures of the programs look like, that is, what are their names, their parameter types, and their return values types if any? The parameters are given by sentences of form

proc: parameter 1 type … parameter n type --> return value type

2) Which sequences of procedure calls are legal, which called *traces* did not considered as being error and given by assertions of the form

L (*trace).*

3) What is the output of legal that ends in a function call, the output value denoted by *V(trace).*

As an example, consider the following specification of a stack module that permits multiple pushes and pops. The module contains three procedures as follows:

- *PUSH* takes an arbitrary positive number of some, as of yet, undetermined type of object as parameters but returns no value;
- *POP* either takes either no parameter or a positive integer as a parameter, but returns no value; and

- *TOP* takes no parameters but returns an object. The syntax of the module is specified thus.

PUSH: obj…obj

POP: [int]

TOP: --> obj

The syntax for *PUSH* is specified by a schema that represents an infinite number of sentences:

PUSH: obj, PUSH: obj obj, etc. Each one of these sentences describes a legal call on PUSH. A programmer is free to treat PUSH as a single procedure that takes an unspecified number of parameters or as a set of procedures, each of which takes a different number of parameters from the rest. Similarly, POP is specified by a schema that represents two sentences: POP: and POP: int. The semantics of the module consists of eight assertions describing the module's behavior:

- If a series of procedure calls has not resulted in an error, then PUSH can be legally called with any object parameter;
- Calling TOP does not result in an error if and only if calling POP does not;
- Calling PUSH with multiple parameters is equivalent to calling PUSH with each parameter individually, leftmost first;
- POP (1) is equivalent to POP;
- Calling POP with n as a parameter is equivalent to calling POP n times;
- Calling PUSH followed by POP does not affect the future behavior of the module;
- If TOP can be legally called, then calling it does not affect the future behavior of the module; and
- The value of any legal series of procedure calls ending in PUSH followed by TOP is the parameter of the last PUSH.

These assertions are symbolized as axioms in an extension of predicate calculus.

(1) L (T) --> L (T.PUSH (o))

(2) L (T.TOP) <--> L (T.POP)

(3) T.PUSH (o_1, . . . , o_n) □ T.PUSH (o_1).PUSH (o_2, . . . ,o_n)

(4) T.POP □ T.POP (1)

(5) i >1 --> T.POP (i) □ T.POP.POP (i −1)

(6) T □T.PUSH (o).POP

(7) L (T.TOP) --> T □ T.TOP

(8) L (T) --> V (T.PUSH (o).TOP) = o

As an example of how to use the specification to make inferences about an implementation's behavior, consider the trace PUSH (tom, jerry).POP.TOP. Substituting the empty trace for T in assertion (3) allows us to conclude that PUSH (tom, jerry).POP.TOP is equivalent to the sequence PUSH (tom).PUSH

(jerry).POP.TOP. Using assertion (6), we can conclude that PUSH (tom).PUSH (jerry).POP is equivalent to PUSH (tom), and hence, that the original trace is equivalent to PUSH (tom).TOP. Using assertion (8) and the assumption that the empty trace is always legal, we can conclude that the original trace returns the value tom. The idea inference of implementation's behavior helps us in simplification processes when we want to validate specifications in abstract data types.

3.3.5 SATISFACTION OF A SPECIFICATION

It must be possible to demonstrate that the implementation satisfies the specification a first approach is to progressively refine the specification until an implementation is reached. If it is possible mathematically to prove that each refinement satisfies the specification, we say that the development process is formal. Another approach is to test the implementation. Test cases are derived from the specification. The results obtained by running the implementation for these test cases are compared with the results prescribed by the specification. Such a development process is said to be informal. For most practical applications, it is not feasible to exhaustively test a system.

From a theoretical viewpoint, proving the correctness of an implementation is more appropriate than testing it. From a practical viewpoint, testing is easier to achieve. We know the strengths and the limitations of formal development processes. For more than

30 years now, computer scientists have investigated the application of mathematics to the development of software systems, with the ultimate goal of developing techniques to prove that an implementation satisfies a specification. Progress has been made, but much remains to be done [37].

The specification phase of the software lifecycle has always played a central role in software engineering, not only because of its intrinsic technical interest, but also because it is critical in determining the success or failure of software products. Our issue is to adopting a specification to satisfy great points of conflicting criteria [2].

3.4 VALIDATION OF A SPECIFICATION

3.4.1 Validation concept

Validation is a means of ensuring that requirements in a scale specify what are supposed to be specified. Even the simplest of validation activities can improve the quality of a draft standard but a well-planned [38], and systematic validation process will identify many technical inaccuracies and completeness that might otherwise have been remained in the specified document. A fundamental issue is to make sure that the specification "matches" the client's needs. This activity is called validation.

Note that we use the verb "match" instead of a stronger verb like "prove", or "demonstrate", in the definition of the validation concept. By its very nature, a specification cannot be "proved" to match the client's requirements. If such a proof existed, then it would require another description of the requirements. If such a description is available, then it is a specification.

A specification is the starting point of the development process. It has the same status as axioms of a mathematical theory. They are assumed to be right. Of course, one can prove that a specification is consistent (i.e., that it does not include a contradiction), just as one can prove that the axioms of a theory are consistent. But this is a different issue from validation.

Validation consists essentially of stating properties about the specification, and proving that the specification satisfies these properties. Properties describe usage scenarios at various levels of abstraction. They can refer to concrete sequences of events, or they can be general statements about the safety or the live ness of the system.

The more properties are stated, the more the confidence in the specification validity is increased. Properties are like theorems of a theory: they must follow from the specification. In summary, validation is an empirical process; a specification is deemed valid until one finds a desired property that is not satisfied [37]

3.4.2 Abstract Data Types ADTs

Data abstraction is played an important role in a software development; the idea of an abstract data type is quite simple, it is a set of objects and operations on those objects. The specification of those operations is defining an interface between abstract data type and the rest of the program [39]. The study of abstract data types led to deeper understanding of types general.

Abstract Data Types are a theoretical concept in computer science, used in the design and analysis of algorithms, data structures, and software systems. An abstract data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects.

There are no standard conventions for defining them. We propose a specification model that captures the behavior of software systems that have an internal state. Abstract Data Types (ADT’s) are a typical software product that has an internal state, but other types of software products fall in this category as well. Formal precise specifications are required and must be completed to informal ones for the reason of formal validation technique [40] to determine if the specifications are meeting the desired features such as completeness and minimality.

3.5 SUMMARY

In this chapter we have introduce specification concepts, formal specification, specification methods and present two types of specifications are model-based specification and behavioral specification. It also introduces some of specification language that introduced before such as Z, B, and Alloy. Two examples of square root and queue were explained in Z specification language. Parnas trace specification is also introduced supported by an example which is closed to our specification. In addition, it gives notion of how to validate specification. This is clearer in validation of stack in the latest of chapter four in section 4.3.10. Abstract data type concepts are also explained in this chapter which are considered one of the most important foundations of our study.

CHAPTER FOUR
MATHEMATICAL BACKGROUND

4.1 INTRODUCTION

This chapter defines the sound of specification; explains the concepts of the algebra of relations, including operation and properties, and explains how the relation support specification; the significance of relations and its role in specification generation; the concept of refinement, its significance; and its role in specification validation; the relational specification of that maintain an internal state of systems; the axiomatic representation of the relational specification of the that maintain the state of systems; and generation and validation of specifications.

4.2 A DISCIPLINE OF SPECIFICATION

The specification of s software product is a description of the properties that the product must have to fulfil its purpose. The specification is usually derived by identifying all relevant stakeholders of the desired software product, eliciting the requirements that they expect the product to meet, formulating and combining these requirements, and compiling them into a cohesive document. While specifications typically pertain to functional and operational requirements, we focus primarily on functional requirements that pertain to the input / output behavior of the software product.

As a product, a specification must meet two conditions:

- Formality. The specification must be represented in such way as to describe precisely what functional behavior is required.
- Abstraction. The specification must describe what requirements the software product must satisfy, not how to satisfy them. In other words, it must focus on WHAT not HOW, HOW is being the prerogative of the designer latter on.

As a process of the identifying stakeholders, eliciting the requirements, combining them, etc. a specification must meet two criteria:

- Completeness: The specification must capture all the relevant requirements of the product.
- Minimality: The specification must capture nothing but the relevant requirements of the product.

  A specification that is deemed to be complete and minimal is said to be valid.

4.3 RELATIONAL MATHEMATIC

4.3.1 Set and relation

In order to specify ADTs we will use sets and relation. We represent sets using a programming-like notation, by introducing variable names and associated data type (sets of values). For example, if we represent set S by the variable declarations ***x: X; y: Y; z: Z,***

Then S is the Cartesian product $XxYxZ$. Elements of S are denoted in lower case s, and are triplets of elements of X, Y, and Z. Given an element s of S, we represent its Xcomponent by *x(s),* its Y-component by *y(s),* and its Z-component by *z(s).* A relation on S is a subset of the Cartesian product *SxS*; given a pair *(s, s')* in *R*, we say that *s'* is an image of *s* by *R*. Special relations on *S* include the universal relation *L= SxS*, the identity relation I= {(s, s')| s'=s}, and the empty relation ϕ = *{}*. To represent relations graphically, we use the Cartesian plane in which set S is represented on the abscissas and the ordinates. Using this device, we represent below an arbitrary relation on S, as

well as L, I, and ϕ as shown in figure 4.1.

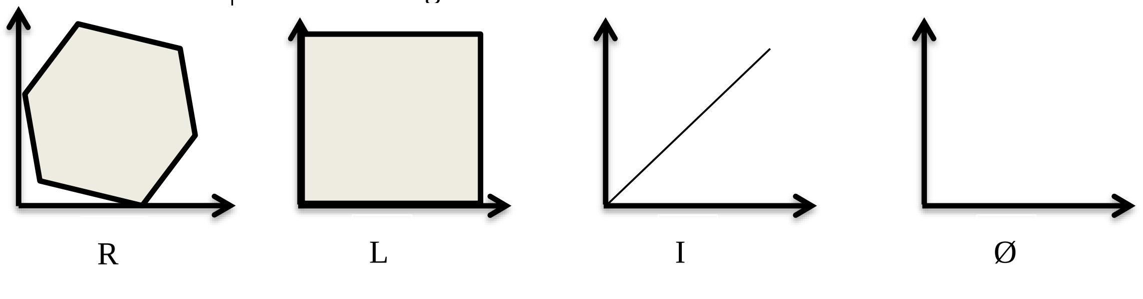


Figure 4.1: S subset of relation on S: Pair(R) relation, Universal (L) relation, Identity (I) relation, and Empty relation.

4.3.2 Operation on relation

Because a relation is a set, we can apply to relations all the operations that are applicable to sets, such as union, intersection, difference, and complement. In addition, we define the following operations:

The converse of relation *R* is the relation denoted by R̂ and define by: R̂ = {(s, s')| (s, s') ϵ R}.

The domain of relation *R* is the subset of *S* denoted by *dom(R)* and define by: *dom(R)*| = {s ∃ s': (s, s') ∈ *R*}.

The range of relation *R* is the subset of *S* denoted by *rng(S)* and defined as the domain of *R*.

The pre-restriction of *R* to (sub) set *A* is the relation denoted by $\boldsymbol{A} \setminus R$ and

defined by $\boldsymbol{A} \setminus R = \{(s, s') \mid s \in A \wedge (s, s') \in R\}$.

The post-restriction of *R* to (sub) set *A* is the relation denoted by $^{\boldsymbol{R}}/_{\boldsymbol{A}}$ = {(s, s') | (s,

s') ∈R s′ ∈ *A*}.

The following figure 4.2 depicts a graphic illustration of a relation R, its complement Ṝ, its converse R̂ respectively.

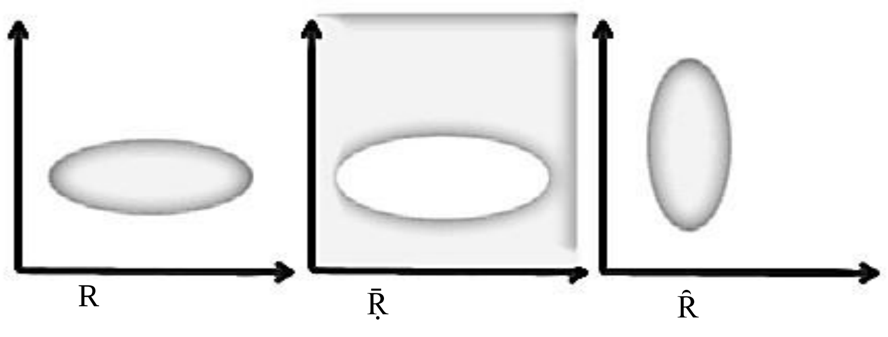


Figure 4.2: Relation, complement and inverse

Given a set A (subset of s), we define three relations of interest:

- The vector defined by A is the relation A×S,
- The inverse vector defined by A is the relation S×A,
- The monotype defined A is the relation denoted by I(A) and defined by: *I(A)*

= {(s, s') | s ∈ *A* ∧ *s′=s*}.

The following figure 4.3 represents, for set A (subset of S), the vector, inverse vector, and monotype defined by A.

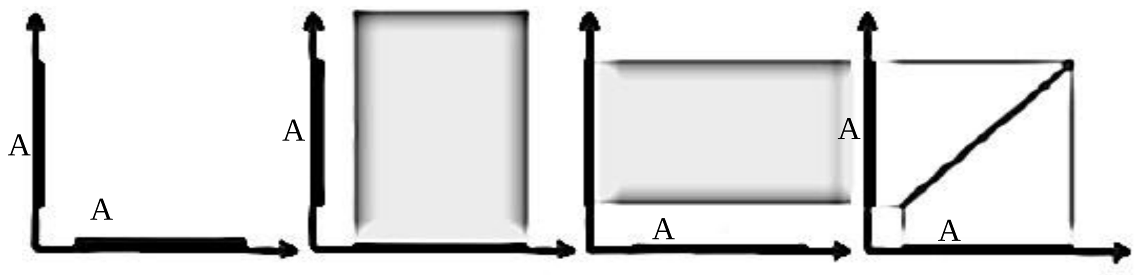


Figure 4.3: (A), Vector(A), Inverse Vector(A), and Monotype(A) respectively.

Given two relations R and R', we let the product of r by R' be denoted by R∗R' (or

RR', if no ambiguity arises) and defined by: *R∗R′* = {(s, s') | ∃s′′:(s, s'') ∈*R* ∧ (s'', s') ∈*R′* }. The figure 4.4 below illustrates the definition of relational product.

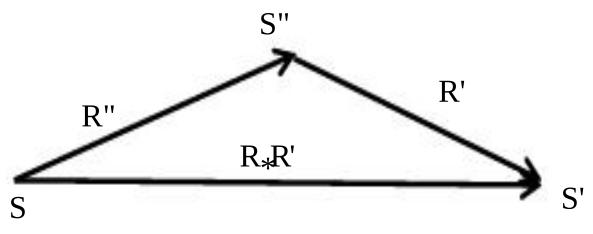


*Figure 4.4: Relational Product of R*R'.*

If we denote the vector and the inverse vector defined by A by, respectively, $\omega(A)$ and $\mu(A)$, then the following identities hold, by virtue of the relevant definition.

- ω(A) = I(A)∗L,
- μ(A) = L∗I(A),
- $\hat{\omega}(A) = \mu$ *(A),*
- $I(A) = \omega(A) \cap I = \mu(A) \cap I$.

Vectors are a convenient (relational) way to represent sets, when we want everything to be a relation. Hence, for example, the domain of relation R can be represented by the vector RL, and the range of relation R can be represented by the inverse vectors LR as shown in figure 4.5.

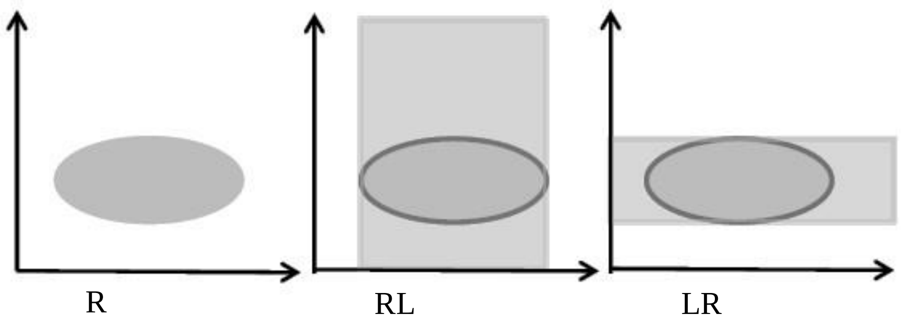


Figure 4.5: Relation R, Vector RL, and Inverse Vector LR

We can represent the pre-restriction and the post-restriction of the relation to a set, say A, using the vector and inverse vector defined by A, as shown in figure 4.6 below.

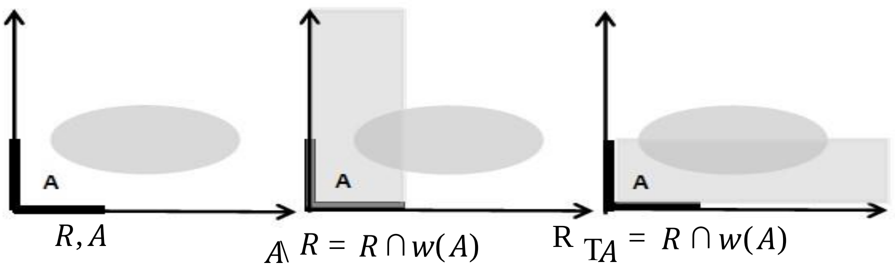

A
A
A
R, A
$A\backslash R = R \cap w(A)$
$R \restriction A = R \cap w(A)$

Figure 4.6: the pre-restriction and the post-restriction of the relation to a set

4.3.3 Properties of Relations

Among the properties of relations, we cite the following:

- A relation R is said to be *total* if and only if $RL=L$.
- A relation R is said to be *surjective* if and only if $LR=L$.
- A relation R is said to be *deterministic* if and only if $RL=L$.
- A relation R is said to be *reflexive* if and only if $I \subseteq R$.
- A relation R is said to be *symmetric* if and only if $R \subseteq \breve{R}$. • A relation R is said to be *transitive* if and only if $RR \subseteq R$.
  - A relation R is said to be *antisymmetric* if and only if $R \cap \breve{R} \subseteq I$.
  - A relation R is said to be *asymmetric* if and only if $R \cap \breve{R} \subseteq \emptyset$.
  - A relation R is said to be *connected* if and only if $R \cap \breve{R} \subseteq L$.
  - A relation R is said to be an *equivalence relation* if and only if it is reflexive, symmetric and transitive.
  - A relation R is said to be a *partial ordering* if and only if it is reflexive, antisymmetric and transitive.
  - A relation R is said to be a *total ordering* if and only if it is partial ordering, and is connected.

Figure 4.7, figure 4.8, and figure 4.9 are illustrating some of these properties.

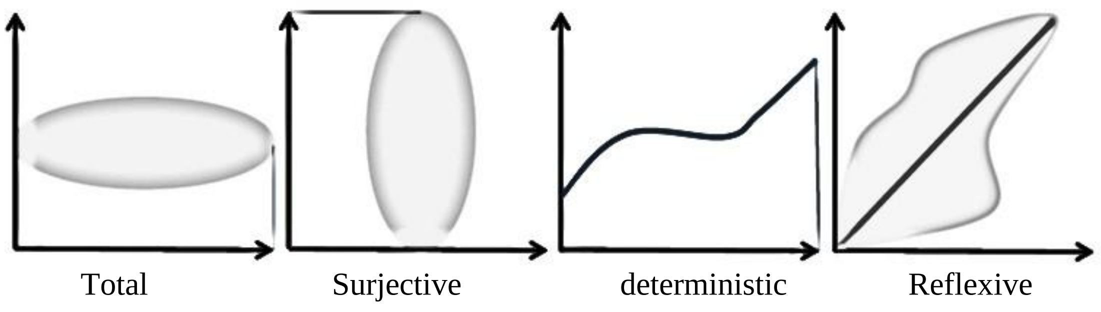


Figure 4.7: Some properties of relation: total, surjective, deterministic, and reflexive respectively.

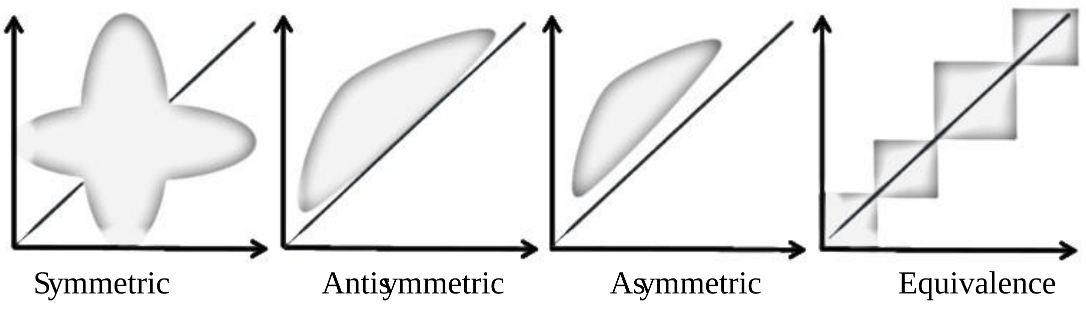


Figure 4.8: Some properties of relation: symmetric, antisymmetric, asymmetric, and equivalence respectively [9].

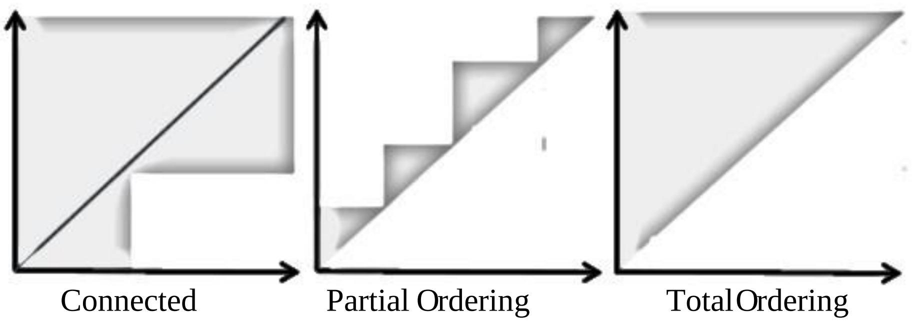


Figure 4.9: Some properties of relation: connected, partial ordering, and total ordering respectively.

4.3.4 Simple input output program

In this section we state that the study of relations have important manner in software specification, validation and testing.

4.3.4.1 Representing Specifications

If one asks CS students in a programming course to write C++ function that reads a real number and compute its square root, they would rush immediately to their computer to write code and run it.; and yet this problem statement may be interpreted in a wide range of manners, leading to a wide range of possible specifications, where space S is defined to be the set of real numbers:

Only non-negative arguments will be submitted; the output is a (positive or nonpositive) square root of the input value: $R_1 = \{(s, s')| \ s \geq 0 \land s'^2 = s\}$.

Only non-negative arguments will be submitted; the output is non-negative square root of the input value: $R_2 = \{(s, s')| \ s \geq 0 \land s'^2 = s \land s' \geq 0\}$.

Only non-negative arguments will be submitted; the output is an approximation (within a precision ε) of a (positive or non-negative) square root of the input value: $R_3 = \{(s, s')| \ s \geq 0 \land |s'^2 = s| < \varepsilon\}$.
Only non-negative arguments will be submitted; the output is an approximation (within a precision ε) of the non-negative square root of the input value:

$R_4 = \{(s, s')| \ s \geq 0 \land |s'^2 = s| < \varepsilon \land s' \geq 0\}$.

Negative arguments may also be submitted; for negative arguments, the output is -1; for non-negative arguments, the output is a (positive or non-positive) square root of the output value: $R_5 = \{(s, s')| \ s \geq 0 \land |s'^2 = s\} \cup \{(s, s'| \ s < 0 \land s' = -1\}$.

Negative arguments may also be submitted; for negative arguments, the output is -1; for non-negative arguments, the output is the non-negative square root of the input value: $R_6 = \{(s, s')| \ s \geq 0 \land s'^2 = s \land s' \geq 0\} \cup \{(s, s')|s < 0 \land s' = -1\}$.

Negative arguments may also be submitted; for negative arguments, the output is arbitrary; for non-negative arguments, the output is approximation (within a precision ε) of a (positive or non-positive) square root of the output value:

$R_7 = \{(s, s')| \ s \geq 0 \land s'^2 = s| < \varepsilon\} \cup \{(s, s')|s < 0\}$.

Negative arguments may also be submitted; for negative arguments, the output is arbitrary; for non-negative arguments, the output is approximation (within a precision ε) of the non-positive square root of the output value:

$R_8 = \{(s, s') | s \geq 0 \wedge |s'^2 = s| < \varepsilon \wedge s' \geq 0\} \cup \{(s, s')|s < 0\}.$

Only non-negative arguments will be submitted; the output must be within ε of the exact square root of the input (comparison with specification R_4: Precision ε applies to the square root scale, rather than the square scale):

$$R^9 = \left\{\{(s, s') | s \geq 0 \ \wedge |s' - \sqrt{s}| < \ \varepsilon\right\}.$$

We could go on and on. This simple example highlights two lessons: First, the importance of precision in specifying program requirements, second, the premise that relations enable us to achieve the required precision.

As a second illustrative example, consider the following requirement pertaining to spaces S defined by and array *a[1...N]* of some type, where N is greater than or equal to 1, a variable x of the same type, and an index variable k, which we use to address array a: (search x in a and place its index in k). again, this simple requirement lends itself to wide range of interpretations, some of which we write below, along with their relational representation:

- Variable x is known to be in a; place in k an index where x occurs in a.

$F_1 = \{(s, s')|(\exists h : 1 \leq h \ \leq N\text{: } a[h\ ] = x) \wedge a[k'] = x\}.$

- Variable x is known to be in a; place in k the first (smallest) index where x occurs in a.
- $F_2 = \{(s, s')|(\exists h : 1 \leq h \ \leq N\text{: } a[h\ ] = x) \wedge a[k'] = x \wedge (\forall h : 1 \leq \ h \ < k'\text{: } a[h\ ] \neq x)\}$

$$= F_1 \cap = \{(s, s')|(\forall h : 1 \leq h \ < k'\text{: } a[h\ ] \neq x)\}.$$

1. Variable x is known to be in a; place in k an index where x occurs in a, while preserving a and x.

$F_3 = F_1 \cap \{(s, s')|a' = a \wedge x' = x\}.$

2. Variable *x* is known to be in a; place in *k* the first (smallest) index where *x* occurs in a, while preserving a and *x*.

$F_4 = F_2 \cap \{(s, s')|a' = a \wedge x' = x\}$.

3. Variable x is not known to be in a; if it is not, place 0 in *k*; else place in *k* an index where *x* occurs in a.

$F_5 = F_1 \cup \{(s, s')|(\forall h : 1 \leq h \leq N: a[h] \neq x) \wedge k' = 0\}$.

4. Variable x is not known to be in a; if it is not, place 0 in *k*; else place in *k* the first (smallest) index where *x* occurs in a.

$F_6 = F_2 \cup \{(s, s')|(\forall h : 1 \leq h \leq N: a[h] \neq x \wedge k' = 0\}$.

5. Variable x is not known to be in a; if it is not, place 0 in *k*; else place in *k* an index where *x* occurs in a, while preserving a and *x*.

$F_7 = F_3 \cup \{(s, s')|(\forall h : 1 \leq h \leq N: a[h] \neq x \wedge k' = 0\}$.

6. Variable x is not known to be in a; if it is not, place 0 in *k*; else place in *k* the first (smallest) index where *x* occurs in a, while preserving a and *x*.

$F_8 = F_4 \cup \{(s, s')|(\forall h : 1 \leq h \leq N: a[h] \neq x \wedge k' = 0\}$.

We note that F_1 can be written simply as $F_4 = \{|s, s'|a[k'] = x\}$, since the clause $(\exists h$: 1 $\leq h \leq N$: $a[h] = x)$ is a logical consequence of $a[k'] = x$. We draw the researcher's attention to the importance of carefully watching which variables are primed and which are unprimed in specification. By written F_1 as we did, we mean that the final value of *k* points to a location in the original array *a* where the original value of *x* is located. As we written, this relation specifies a search program. If, instead of F_1, we had written the specification as follows:

$$F'_1 = \{(s, s')|a[k'] = x\},$$

Then it would be possible to satisfy this specification by the following simple program:

{k=1; x=a[1];}.

If, instead of F_1, we had written the specification as:

$$F''_1 = \{(s, s')|a'[k'] = x\},$$

Then it would be possible to satisfy this specification by the following simple program:

{k=1; a[1]= x;}.

If, instead of F_1, we had written the specification as:

$F'''_1 = \{(s, s')|a'[k'] = x'\}$,

Then it would be possible to satisfy this specification by the following simple program:

{k=1*;* x=0*;* a[1]= 0;}.

None of these three programs is performing a search of variable *x* in array a.

4.3.4.2 Ordering Specifications

When we consider specifications on a given space S, we find it natural to order them according to the strength of their requirement, i.e. some of them impose more requirements than others. Let us, for the sake of illustration, consider the specifications of the program written in the previous section:

a) $F_1 = \{(s, s')|(\exists h : 1 \leq h \ \leq N: a[h\ ] = x) \wedge a[k'] = x\}$.
b) $F_2 = F_1 \cap = \{(s, s')|(\forall h : 1 \leq h \ < k': a[h\ ] \neq x)$
c) $F_3 = F_1 \cap \{(s, s')|a' = a \wedge x' = x\}$
d) $F_4 = F_2 \cap \{(s, s')|a' = a \wedge x' = x\}$
e) $F_5 = F_1 \cup \{(s, s')|(\forall h : 1 \leq h \ \leq N: a[h\ ] \neq x) \wedge k' = 0\}$
f) $F_6 = F_2 \cup \{(s, s')|(\forall h : 1 \leq h \ \leq N: a[h\ ] \neq x \wedge k' = 0\}$
g) $F_7 = F_3 \cup \{(s, s')|(\forall h : 1 \leq h \ \leq N: a[h\ ] \neq x \wedge k' = 0\}$
h) $F_8 = F_4 \cup \{(s, s')|(\forall h : 1 \leq h \ \leq N: a[h\ ] \neq x \wedge k' = 0\}$

It is natural to consider that F_2 is stronger that F_1 since the latter would be satisfied with k' pointing to any occurrence of the x in a, while the former requires that k' point to the smallest such occurrence. Also, it is natural consider that F_3 is stronger than F_1 since the latter requires that a and x be preserved whereas the former does not; for the same reason, F_4 is stronger than F_2. On the other hand, F_5 can be considered stronger than F_1, since the latter makes provisions for the case when x is not in a, whereas the former does not; for the same reason, we can consider that

F_6 is stronger than F_2, that F_7 is stronger than F_3, and that F_8 is stronger than F_4.

Figure 4.10 is depicted ordering of these relations.

We notice that F_2 and F_6 are both considered stronger than F_1, but while former is a subset of F_1, the latter is a superset thereof. There appears to be two non-exclusive ways for specification R to be considers stronger than a specification R': by having a lager domain, and by having fewer images for elements in the common domain.

Whence the following definition.

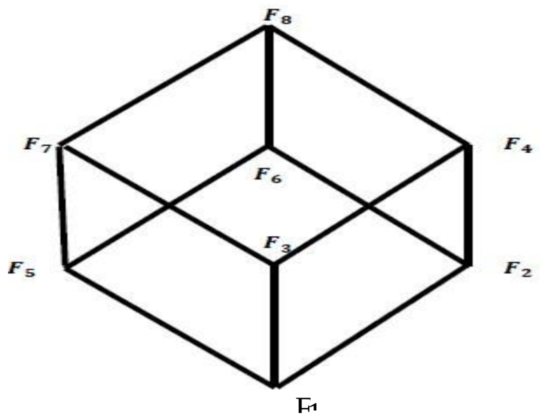


Figure 4.10: order of the relations.

Definition: **refinement**. Given a set S and two relations R and R' on S. we say that R refines R' if and only if R has a larger domain than R' and has fewer image for elements in the domain of R'. Formally, $RL \supseteq R'L$ and $R'L \cap R \subseteq R'$. Henceforth, we use the term refines to refer to the property of being a stronger specification. We admit without proof that the relation refines is a partial ordering between specifications, i.e. that it is reflexive shows two relations, R' and R'', that refine relation R; figure 4.11 shows these relations.

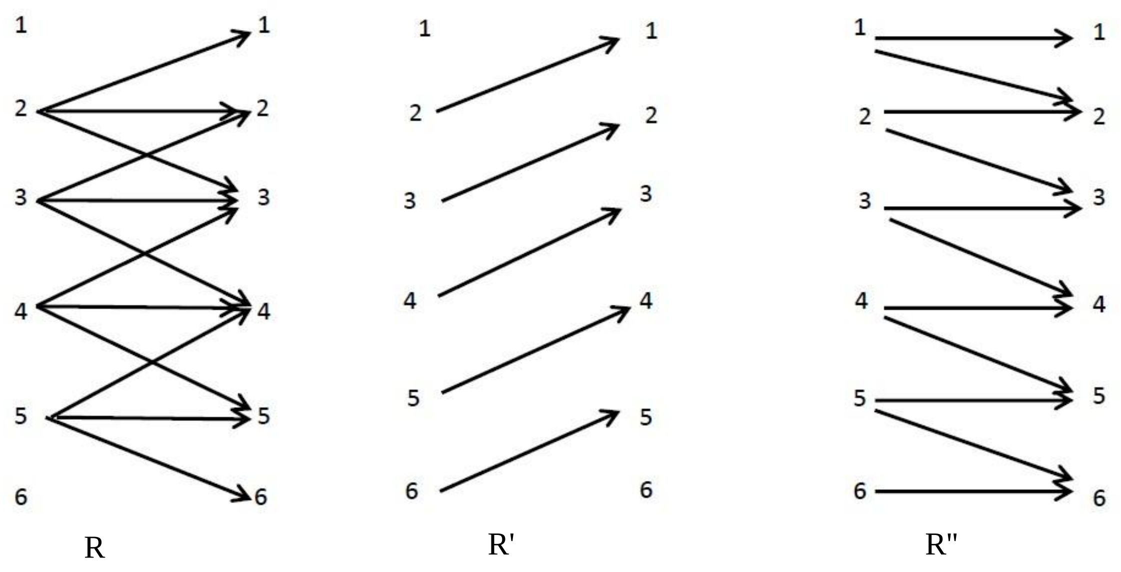


Figure 4.11: Relation R, refine relation R' and R''

4.3.5 Specifications Generation

Let space *S* be defined by a real array *a[1…N]* and a real variable *x* and index variable *k*. we are interested to write a relation to reflect the following requirement:

Place in *x* the largest value of *a* and in *k* the smallest index where the largest value occurs. For example, if array *a* has the following value shown in table 4.1 below:

Table 4.1: An array of 12 values.

1	2	3	4	5	6	7	8	9	10	11	12
8.8	9.0	5.1	9.3	9.3	0.02	4.2	8.1	9.3	3.0	2.5	9.3

Then we want *x'* to be equal to 9.3 and *k'* be equal to 4. Because it is too difficult to specify all the requirements at once, we consider them one by one:

Place in *x* a value larger than all the values in array:

$$M_1 = \{(s, s') | \forall h : 1 \le h \le N : x' \ge a[h]\}.$$

Place in x element of the array (note that M_1 alone ensure that x' is greater than all the elements of array, but does not ensure that it is the max: 500.0 could be a possible values, for the array above):

$$M_2 = \{(s, s')|\exists h : 1 \leq h \leq N: x' \geq a[h]\}.$$

Place in k' *an* index of a where the maximum of the array occurs:

$$M_3 = \{(s, s')|\ x' = a[k']\}.$$

Ensure that no index smaller than k' carries the maximum of the array (hence ensuring that k' is the smallest index):

$$M_4 = \{(s, s')|\forall h : 1 \leq h < k': x' \neq a[h]\}.$$

Then we compute the overall specification as the intersection of M_1, M_2, M_3, *and* M_4 i.e. $M = M_1 \cap M_2 \cap M_3 \cap M_4$.

As a second example, we consider spaces S made up of three non-negative variables *x, y, z,* and a variable *t* that represents the enumerated type: {scalene, isosceles, equilateral, rightisoceles, right}. We assume that the label isosceles is reserved for triangles that are isosceles but not equilateral, and that the label isosceles is reserved for triangles that are right but not isosceles.

To write this specification, we write one relation for each type of triangle, then from their union. To this effect, we define the following predicates in triplets of real numbers:

- $Equi(x, y, z) \equiv (x = y \wedge y = z)$.
- $Iso(x, y, z) \equiv (x = y \vee y = z \vee x = z)$.
- $Right(x, y, z) \equiv (x^2 = y^2 + z^2 \vee y^2 = x^2 + z^2 \vee z^2 = x^2 + y^2)$.

Using these predicates, we define the following relations:

$T_1 = \{(s, s')|\ Equi(x, y, z) \wedge t' = equilateral\}$.

$T_2 = \{(s, s')|\ Iso(x, y, z) \wedge \neg Equi(x, y, z) \wedge \neg Right(x, y, z) \wedge t' = isoceles\}$.

$T_3 = \{(s, s')|\ Iso(x, y, z) \wedge (x, y, z) \wedge t' = righ\ tisoceles\}$.

$T_4 = \{(s, s')|\ Righ\ t(x, y, z) \wedge \neg Iso(x, y, z) \wedge t' = righ\ t\}$.

$T_5 = \{(s, s')|\ \neg Iso(x, y, z) \wedge \neg Equi(x, y, z) \wedge \neg Righ\ t(x, y, z) \wedge t' = scalene\}$.

Using these relations, we form the relational specification of the triangle classification problem:

$T = T_1 \cup T_2 \cup T_3 \cup T_4 \cup T_5$.

From these two examples, we want to discuss the question of how do we generate a complex specification from simple elementary specifications?

In the case of the specification that finds the maximum of the array, we compute the overall specification as the intersection of elementary specifications; in the case of the specification of triangle classification, we compute the overall specification as the union of elementary specifications.

It appears that we use the intersection when the domains of the elementary specifications are identical; and we use the union when the domains of the elementary specifications are disjoint. In the former case we generate the compound specification as the conjunction of elementary properties; whereas in the latter case we generate the compound specification by case analysis.

The question that we wish to address then is: giver two relations *R1* and *R2* on space S, how do we compose them into a specification that captures all the requirements of *R1* and all the requirements of *R2* for the sake of completeness and for the sake of minimality, assuming that the domains of *R1* and *R2* are neither necessarily identical nor necessarily disjoint? Consider the figure 4.12 below which depicting the configuration of *dom(R1)* and *dom(R2).*

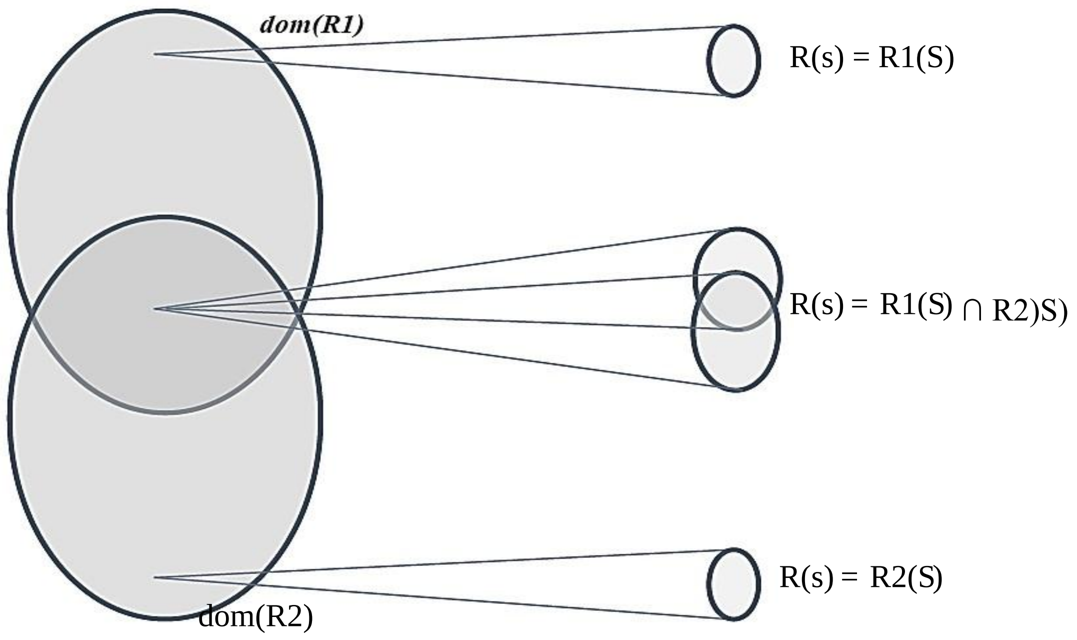


Figure 4.12: Relation of R1 and R2 and their intersection.

If we want R to capture all the specification information of R1 and all the specification information of R2, then R has to be identical to R1 outside the domain of R2, and identical to R2 outside the domain of R1, and for each element of the intersection of the domains of R1 and R2, it has to be identical to the intersection of R1 and R2. This justifies the following definition.

Definition. The join of two relations R_1 and R_2 on set S is denoted by $R_1 \oplus R_2$ and defined by: $R_1 \oplus R_2 = \overline{\overline{R_2}L} \cap R_1 \cup \overline{\overline{R_1}L} \cap R_2 \cup (R_1 \cap R_2)$.

This formula is a mere relational representation of the figure 4.12 above, depicting how $R = R_1 \oplus R_2$ can be derived from R_1 and R_2. The following proposition, gives an important property of the *join* operator.

Proposition. Let R_1 and R_2 be two relations on set S. if R_1 and R_2 satisfy the following condition, $R_1 L \cap R_2 L = (R_1 \cap R_2)L$,

Which we call the compatibility condition, then $R_1 \oplus R_2$ is the least refined relation that refines $R_1$and R_2 simultaneously. If $R_1$and R_2 do not satisfy the compatibility condition, then there exists no relation that refines them both, Figure 4.13 shows an example of two relations $R_1$and R_2 that satisfy compatibility condition, and shows their join.

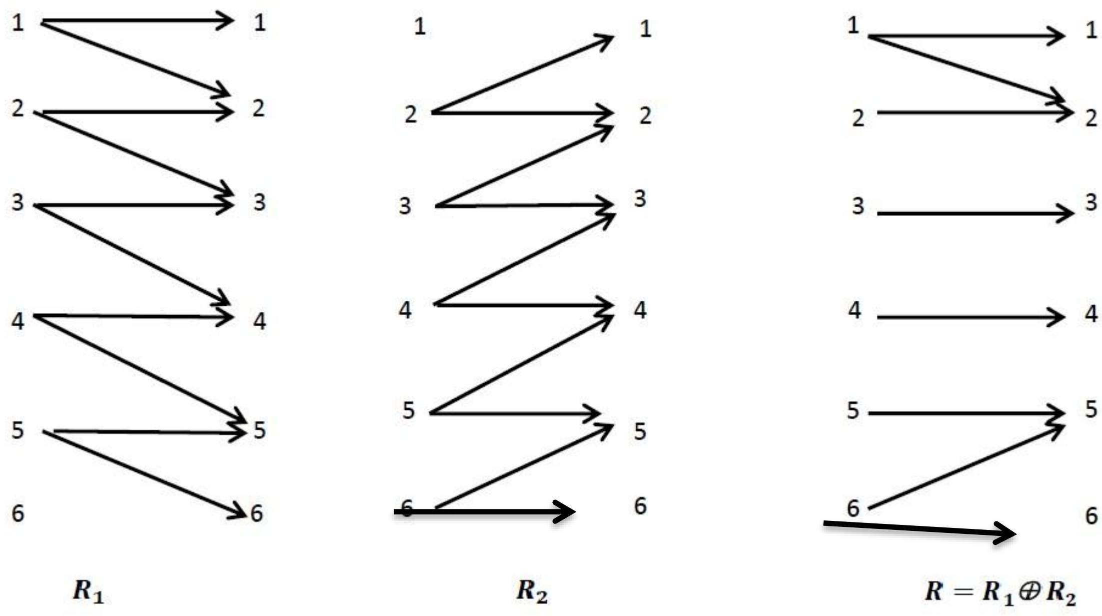


Figure 4.13: Relation R1 and R2 and their compatibility condition.

On input 1 (outside the domain of R_2), R behaves as R_1; on input 6 (outside the domain of R_1), R behaves like R_2 and on inputs {2, 3, 4, 5} (the intersection of the domains of R_1 and R_2), R behaves like the intersection of R_1 and R_2 (which includes {(2, 2)(3, 3)(4, 4)(5, 5)}). Consider the following relations R_1 and R_2, as shown in figure 4.14 below:

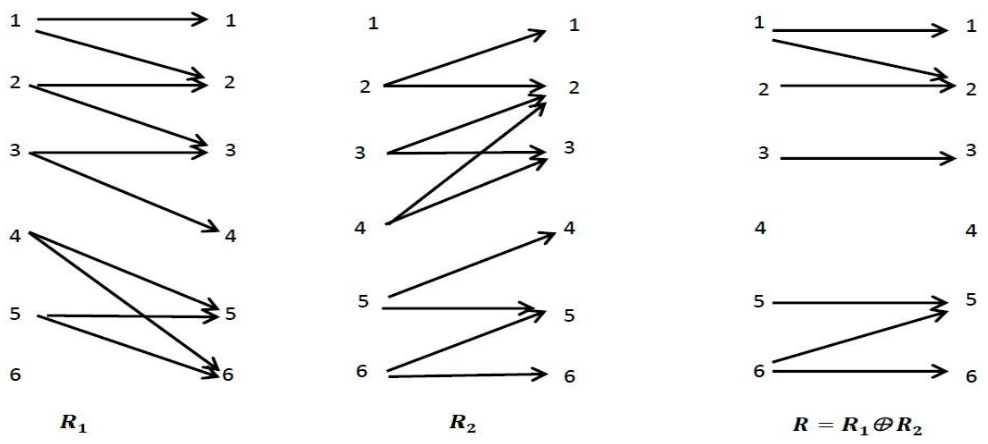


Figure 4.14: Relation R1 and R2 and their intersection.

In this case, R_1 and R_2 do not satisfy the compatibility condition, since 4 belongs to the domain of each one of them but does not belong to the domain of their intersection. Indeed, it is not possible to find a relation that refines them simultaneously, since R_1 assigns images 5 and 6 to 4, where R_2 assigns images 2 and 3; there is no value that R may assign to 4 to satisfy both $R_1$and R_2.

4.3.6 Specification validation

Software product can be validated as a whole, but requirements specification also might be validated before the entire system is finished. Software validation is the process of evaluating software during or at the end of the development process to determine whether it satisfied requirements [2]. However, software engineering literature is replete with example of software projects that fail, not because programmers do not know how to write code or how to test it, but rather because analysts and engineers fail to write valid specifications, i.e. specifications that capture all the relevant requirements that satisfy the completeness and minimality [9]. Consequently, it is important to validate specifications for completeness and minimality, and to invest the necessary resources to this effect before proceeding with subsequent phases of software lifecycle. In this section, the researcher briefly and cursorily discusses the process of specification validation, in the context of the relational specification, with the modest goal of giving the reader some sense of what it may mean to validate a specification.

Let us start with a very simple illustrative example: We consider space S defined by natural variables x and y, and we consider the following requirement:

Increase x whiles preserving the sum of x and y.

We submit the following relations as possible specifications for this requirement:

- ✦ $N_1 = \{(s, s')|x \leq x' \wedge x + y = x' + y'\}$. ✦ $N_2 = \{(s, s')|x + y = x' + y' \wedge y \geq y'\}$.
- ✦ $N_3 = \{(s, s')|x' = x + 3 \wedge y' = y - 3\}$.
- ✦ $N_4 = \{(s, s')|x < x' \wedge y > y'\}$.
- ✦ $N_5 = \{(s, s')|x - x' = y' - y \wedge y > y'\}$.

- ✦ $N_6 = \{(s, s')|x < x' \wedge x - x' = y' - y\}$.
- ✦ $N_7 = \{(s, s')|x' = x + 1 \wedge x + y = x' + y'\}$.
- ✦ $N_8 = \{(s, s')|x + y = x' + y' = \wedge\ y' = y - 2\}$.

We can ask the following questions: which of these specifications is complete; and for those that are complete, which are minimal. Table 4.2 below shows the answers to these questions (if a specification is not complete, it makes no sense to check its minimality):

Table 4.2: Answers of complete and minimality of specification.

Specification	Complete?	Minimal?	Valid?
$N_1 = \{(s, s')|x \leq x' \wedge x + y = x' + y'\}$.	No	N/A	No
$N_2 = \{(s, s')|x + y = x' + y' \wedge y \geq y'\}$.	No	N/A	No
$N_3 = \{(s, s')|x' = x + 3 \wedge y' = y - 3\}$.	Yes	No	No
$N_4 = \{(s, s')|x < x' \wedge y > y'\}$.	No	N/A	No
$N_5 = \{(s, s')|x - x' = y' - y \wedge y > y'\}$.	Yes	Yes	Yes
$N_6 = \{(s, s')|x < x' \wedge x - x' = y' - y\}$.	Yes	Yes	Yes
$N_7 = \{(s, s')|x' = x + 1 \wedge x + y = x' + y'\}$.	Yes	No	No
$N_8 = \{(s, s')|x + y = x' + y' = \wedge\ y' = y - 2\}$.	Yes	No	No

Specification N_5 and N_6 are complete and minimal, they are identical; in fact, they specify that *x* must be increases while preserving the sum of *x* and *y*. Specification N_1 and N_2 are not complete because they do not stipulate that x must increase; they allow it to stay constant; and specification N_4 is not complete because it fails to specify that the sum of x and y must be preserved. Specification N_3, N_7 and N_8 are complete but not minimal because they specify by how much x be increased, which is not preconditioned in the requirement.

In the example above, we wrote the specifications on the basis of the proposed requirement (to increase x while preserving the sum of x and y) and we judged the completeness and minimality of candidate specifications by considering the same source, i.e. the proposed requirement. If the same person or group is tasked with generating the candidate specifications and judging their validity according to completeness and minimality then the same biases that cause the person to write invalid specifications may cause him or her to overlook of the validity of their specification. The only way to ensure a measure of confidence in the validation of the specification is to separate the team that generates the specification from the team that validates it. I am and my colleagues Nahid and Amal take sex abstract data types for example stack, queue, list, set, sequence, and multiset, and work in two teams and two phases. Everybody act as team of specification generation and manually specified two types of ADTs in sound of axiomatic specification; the other two persons act as a team of specification validation and manually validated that two types of ADTs. Table 4.3, shows this effort of our activities:

Table 4.3: Specification generation and validation activities.

Phase	activity	Specification Generation	Specification validation
Specification Generation		Generating the specification from source of requirements	Generating validation data from same source of requirements
Specification validation		Updating the specification according to feedback from the validation team	Testing the specification against the validation data generated above

Notice that the specification team and validation team they work independently for each other For the sake of the redundancy.

The specification generation phase: In the specification generation phase, the specification team generated the specification by referred to all the sources of the requirements. Using the exact same sources, the validation team generated validation data that it intends to test the specification against. Here we can explain that the validation data have two properties that we validate according to which are:

- Completeness properties, these are the properties that the specification must have, but the validation team suspects the specification team may fail to write.
- Minimality properties, these are properties that specification must not have, but the validation team suspects the specification generation team may write inadvertently.

*The specification validation phase***:** int specification validation phase, the validation team had tested the specification against completeness and minimality data generated in previous phase, while the specification team updates the specification, if it turns was not complete or not minimal.

Here we explain and defined the two properties completeness and minimality in, if the reader have question such as what does the validation data take, and how one test a specification against the generated validation data.

Definition: **completeness**. Given a requirements document, a *completeness* property *V* is a relation that represents requirements information that candidate specifications must capture. A specification *R* is said to complete with respect to *V* if and only if refines *V*.

Implicit in this definition is that a good completeness property is one that has the potential to detect an incomplete specification; express in other way, a good completeness property is one that validation team believes the specifier team may have overlooked.

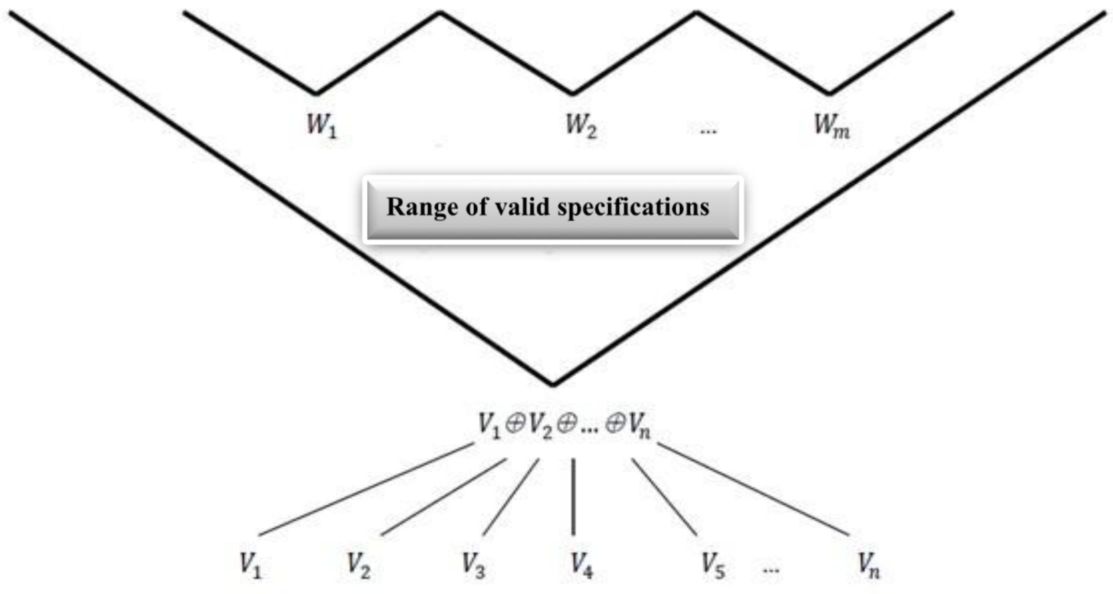
Definition: **Minimality.** Given a requirements document, a *Minimality* property *W* is a relation that represents requirements information that candidate specifications must not capture. A specification *R* is said to minimal with respect to *W* if and only if refines *W*.

Implicit in this definition is that a good minimality property is one that has the potential to detect a non-minimal specification; express in other way, a good minimality property is one that the validation team believes the specifier team may have inadvertently recorded in the specification.

Completeness and minimality are not absolute attributes, but rather relative with respect to selected completeness and minimality properties, as provided in the following definition.

Completeness and minimality are not absolute attributes, but rather relative with respect to selected completeness and minimality properties, as provided in the following definition. **Definition**: **validity.** A specification R is said to be valid with respect to completeness properties $V = \{V_1, V_2, V_3, \dots, V_N\}$ and minimality properties of V and minimal with respect to every element of W.

We admit without proof that if R refines all of $V_1, V_2, V_3, \dots, V_N$ then it refines their join. Hence the range of valid specification with respect to completeness properties $V = \{V_1, V_2, V_3, \dots, V_N\}$, and minimality properties $W = \{W_1, W_2, W_3, \dots, W_N\}$ is represented in the figure 4.15 below:



W_1
W_2
...
W_m
Range of valid specifications
$V_1 \oplus V_2 \oplus \dots \oplus V_n$
V_1
V_2
V_3
V_4
V_5
...
V_n

Figure 4.15: valid specification with respect to properties of completeness and minimality.

As a illustrative example of specification validation, consider the following requirement pertaining to space S defined by an array $a[1 \dots N]$of some types, a variable x of the same type, and index variable k, which we use to address array a: given that x is known to be in *a, place in k the smallest index where x occurs.* This is a variation of the example that we discussed in section 4.3.4.1

4.3.6.1 4.3.4.5 Specification Generation Phase

Example of completeness properties includes the following:

- If each cell of array a contain the index of that cell, and if x=1, then k' should be 1. $V_1 = \{(s, s') | (\forall h : 1 \leq h \leq N: a[h] = h) \land x = 1 \land k' = 1\}$.

- If array a contains 1 everywhere and x =1, then k' should be 1.

  $V_2 = \{(s, s')|(\forall h : 1 \leq h \leq N: a[h] = 1) \wedge x = 1 \wedge k' = 1\}$.
- If array a contains the sequence $1 \ldots N$ in increasing order and if x=N, then k' should be N.

$V_3 = \{(s, s')|(\forall h : 1 \leq h \leq N: a[h] = h) \wedge x = N \wedge k' = N\}$.

Example of minimality properties includes the following:

1. There is no requirement to preserve x.

$$W_1 = \{(s, s')|(\exists h : 1 \leq h \leq N: a[h] = x) \wedge x' = x\}.$$

2. There is no requirement to preserve a.

$$W_2 = \{(s, s')|(\exists h : 1 \leq h \leq N: a[h] = x) \wedge a' = a\}.$$

4.3.6.2 Specification validation phase

So far, we have looked at the requirements documentation, but we have not looked at candidate specifications; generating validation data independently of specification generation is important, for the sake of redundancy. Now, let us consider a candidate specification, and check whether it is complete with respect to the completeness properties, and minimal with respect to the minimality properties. We consider specification F_2, introduced in section 4.3.4.1 as:

$$F_2 = \{(s, s')|(\exists h : 1 \leq h \leq N: a[h] = x) \wedge a[k'] = x \wedge (\forall h : 1 \leq h < k': a[h] \neq x)\}$$

To prove that F_2 refines V_1, we must prove that F_2 has a larger domain than V_1, and that the restriction of F_2 to the domain of V_1 is a subset of V_1. We find,

$F_2L = \{(s, s')|(\exists h : 1 \leq h \leq N: a[h] = x)\}$. $V_1L = \{(s, s')|(\forall h : 1 \leq h \leq N: a[h] = h) \wedge x = 1\}$.

Clearly, V_1L is a subset of F_1L. We compute the restriction of F_2 to V_1L, and we find:

$F_2 \cap V_1L$

= {substitution}

$\{(s, s')|(\forall h : 1 \le h \le N\colon a[h] = h) \wedge x = 1 \wedge (\exists h : 1 \le h \le N\colon a[h] = x) \wedge$

$a[k'] = x \wedge (\forall h : 1 \le h < k'\colon a[h] \neq x)\}$

= {substitution}

$\{(s, s')|(\forall h : 1 \le h \le N\colon a[h] = h) \wedge x = 1 \wedge a[k'] = x \wedge (\forall h : 1 \le h <$

$k'\colon a[h] \neq x)$

= {logic simplification}

$\{(s, s')|(\forall h : 1 \le h \le N\colon a[h] = h) \wedge x = 1 \wedge k' = 1 \wedge (\forall h : 1 \le h < k'\colon a[h] \neq$

$x)\}$

= {logic simplification}

$\{(s, s')|(\forall h : 1 \le h \le N\colon a[h] = h) \wedge x = 1 \wedge k' = 1\}$

= {substitution}

V_1.

We now consider the completeness property V_2. To prove that F_2 refines V_2, we must prove that F_2 has a larger domain than V_2, and that that the restriction of F_2 to the domain of V_2 is subset of V_2. We find,

$F_2L = \{(s, s')|(\exists h : 1 \le h \le N\colon a[h] = x)\}$.
$V_2L = \{(s, s')|(\forall h : 1 \le h \le N\colon a[h] = h) \wedge x = 1\}$.

Clearly, V_2L is a subset of F_2L. We compute the restriction of F_2 to V_2L and we find:

$F_2 \cap V_2L$

= {substitution}

$\{(s, s')|(\forall h : 1 \leq h \leq N: a[h] = h) \wedge x = 1 \wedge (\exists h : 1 \leq h \leq N: a[h] = x) \wedge a[k'] = x \wedge (\forall h : 1 \leq h < k': a[h] \neq x)\}$

= {simplification, redundancy}

$\{(s, s')|(\forall h : 1 \leq h \leq N: a[h] = 1) \wedge 1 \leq k' \leq N \wedge x = 1 \wedge (\forall h : 1 \leq h < k': a[h] \neq 1)\}$

= {logic}

$\{(s, s')|(\forall h : 1 \leq h \leq N: a[h] = h) \wedge x = 1 \wedge k' = 1\}$

= {substitution}

V_2.

We now consider the completeness property V_3. To prove that F_2 refine V_3, we must prove that F_2 has a larger domain than V_3, and that that the restriction of F_2 to the domain of V_3 is a subset of V_3. We find,

$F_2L = \{(s, s')|(\exists h : 1 \leq h \leq N: a[h] = x)\}$.
$V_3L = \{(s, s')|(\forall h : 1 \leq h \leq N: a[h] = h) \wedge x = 1\}$.

Clearly, V_3L is a subset of F_2L. We compute the restriction of F_2 to V_3L, and we find:

$F_2 \cap V_3L$

= {substitution}

$\{(s, s')|(\forall h : 1 \leq h \leq N: a[h] = h) \wedge x = N \wedge (\exists h : 1 \leq h \leq N: a[h] = x) \wedge a[k'] = x \wedge (\forall h : 1 \leq h < k': a[h] \neq x)\}$

= {simplification}

$\{(s, s')|(\forall h : 1 \leq h \leq N: a[h] = h) \wedge x = N\ a[k'] = N \wedge (\forall h : 1 \leq h < k': a[h] \neq x)\}$

= {logic}

$\{(s, s')|(\forall h : 1 \leq h \leq N: a[h] = h) \wedge x = N \wedge k' = N\}$

= {substitution}

V_3.

We turn our attention to checking the minimality of F_2 with respect to $W_1$and W_2.

$F_2 = \{(s, s')|(\exists h : 1 \leq h \leq N: a[h] = x) \wedge a[k'] = x \wedge (\forall h : 1 \leq h < k': a[h] \neq x)\}$

$W_1 = \{(s, s')|(\exists h : 1 \leq h \leq N: a[h] = x) \wedge x' = x\}$.

Because F_2 and W_1 have the same domain, the only way to prove theat F_2 does not refine W_1 is to prove that $F_2 \cap W_1L$ is not a subset of W_1. To this effect, we compute:

$F_2 \cap W_1L$

= { F_2 and W_1 have the same domain}

F_2 .

Which is not a subset of W_1. Hence $F_2$is minimal with respect to W_1. We can prove, likewise, that it is minimal with respect to W_2. Indeed, F_2 does not preserve x nor a.

4.3.7 Reliability and safety

The introduction of the refinement ordering we presented in this chapter enable us to revisit a concepts of software properties such as reliability and safety. The reliability and safety of a system is its likelihood of avoiding failure whereas the safety of a system is its likelihood of avoiding catastrophic failure; because catastrophic failures are failures, one may be tempted to argue that a reliable system is necessarily safe, but that is not the case. Indeed, reliability and safety are not logical properties but stochastic properties; hence the argument that catastrophic failures are failures does not enable us to infer that reliable systems are necessarily safe. Rather, because the stakes attached to meeting the safety requirements are much higher than those attached to meeting the reliability requirement; the threshold of probability that must be reached for a system to be considered safe is much higher than the threshold of probability that must be reliable.

This idea can be elucidated by means of the refinement ordering:

Let *R* be the specification that represents the reliability requirements of a system and let *F* be the specification that represents its safety requirements. For the sake of illustration, we consider a simple example of a system that controls the operation of traffic light at an intersection.

Specification *R* captures the requirements that the traffic light must satisfy in term of how it schedules the green, orange and red light of each incoming street, along with the walk and do not walk signs for pedestrians crossing the street. Such requirements must dictate the sequence of light configurations, (i.e. which streets have green, which streets have orange, which streets have red, which walkways have a walk signal, which walkways have a flashing walk signal, which walkways have a do not signal, etc.), as well as how much each configuration lasts in order to optimize traffic flow, fairness, pedestrian safety, etc.

Specification *F* focuses on two safety critical requirements: First that no orthogonal streets have a green light at the same time; no street has a green light for cars and pedestrians at the same time.

In the following paragraph we explain a typical relationship of both reliability and safety.

The stakes attached to violating a safety requirement are much heavier than the stakes attached to reliability requirement. Violating a reliability requirement may cause a relatively minor inconvenience, such as a traffic jam; by contrast, violating a safety requirement may cause an accident that involves injuries or loss of life.

As a consequence of this difference in stakes, we impose different probability thresholds to the different properties. To consider that a system is reliable, it suffices that it meets the reliability requirements with a probability of 0.99 over a unit of operation time; having traffic jam one percent of the time is acceptable. But to consider that a system is safe, we need a higher probability of meeting the safety requirement; having a fatal accident one percent of the time is not acceptable; a probability threshold of 0.99999 is more palatable.

The reliability requirements specification *R* refines the safety requirement specification *F*, if we consider the sample of traffic lights, and we assume that the requirements specification is valid, and then the reliability requirement clearly subsumes the safety requirement since any behavior that abides by the reliability requirement excludes that two orthogonal streets have a green light simultaneously, or that a street has a green light while at the same time a walkway that crosses it has a walk signal.

It is much easier to prove that a candidate program satisfies a safety requirement *F* than it is to prove that it satisfies the reliability requirement *R*, for the simple reason that a reliability requirement is typically significantly more complicated. Fortunately, because the safety requirement is simpler, we can verify candidate programs against it with greater thoroughness, hence achieve greater confidence reflected a higher probability; that candidate program meets this requirement.

Figure 4.16 below shows specification R and F, ordered by refinement, and illustrates the relationship between the various possible behaviors of candidate program, with corresponding probabilities of the behaviors in question: reliable behavior, (possibly unreliable but fail-safe behavior), and unsafe behavior.

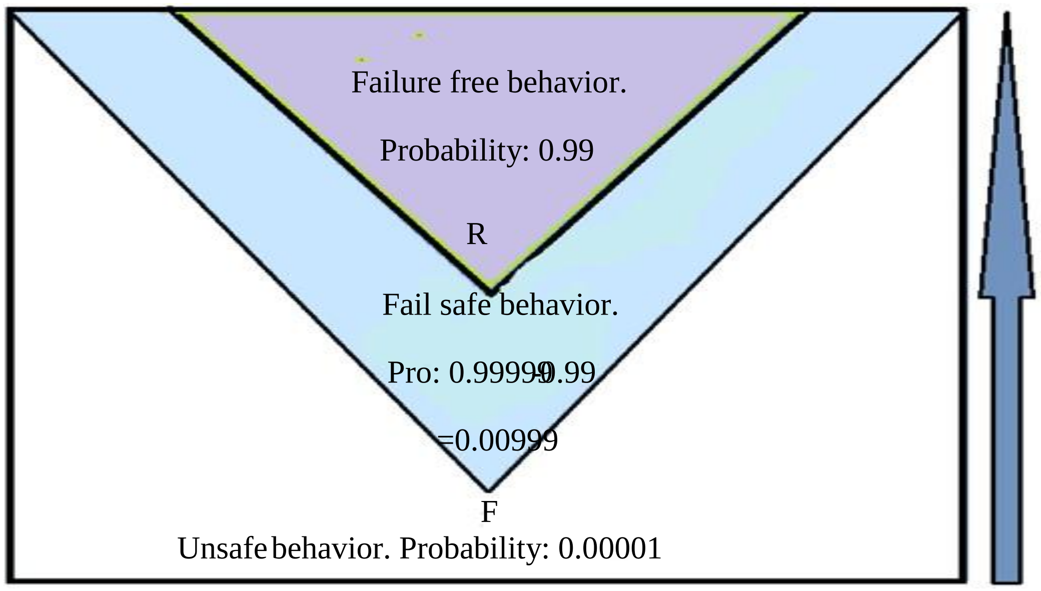


Figure 4.16: the relationship between reliable behavior and unsafe behavior of R and F specification.

4.3.8 State- based systems

Whereas specifications we have studied so far are adequate for specifying programs that take an input as initial state and map in onto output as final state, they are inadequate to represent program whose response depends not only on their input, but their internal state; the subject of this section is to explore ways to specify such systems.

4.3.9 A relational model

Let us recall from our discussion in chapter 1, specification have to have two key attributes, which are formality and abstraction. We can achieve formality by using a mathematical notation, which associates precise semantics to each statement. As for abstraction, we can achieve it by ensuring that the specifications describe the externally observable attributes of candidate software products, but do not specify, dictate or otherwise favour any specific design implementation.

We consider the following description of stack data type: A stack is data type is used to store items through operation push() and to remove them in inverse 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() checks whether the stack has any items stored; operation init() reinitializes the stack to an initial situation. If we want stack, without saying anything about how to implement it; how would we do it? Most data structure courses we studied before introduce stack by showing a data structure made up of an array and index into the array, and by explaining how push and pop operations affect the data structure; but such an approach violates the principle of abstraction since it specifies the stack by describing a possible implementation thereof. An alternative could be to specify the stack by means of an abstract list, along with list operations, without specifying how the list is implemented. We argue that this too violates the principle of abstraction, as it dictates a preferred implementation; in fact, a stack does not necessarily require a list of elements, regardless of how the list is represented.

Consider that a stack which stores identical elements can be implemented by a simple natural number n:

- init():   {n=0;}
- push(a):   {n=n+1;} // a is the only value that can be stacked
- pop():   {if (n>0) {n=n-1;}}
- top():   {if (n>0) {return a;} else {return error;}}
- size():   {return n;}
- empty():   {return (n==0);}
- Consider a stack that stores two possible symbols (e.g. '{' and '}') can also be implemented without any form of list, using a simple natural number n:
- init():   {n=1;}
- push(a):   {n=2*n+ code(a);}

// where code(a) maps the two symbols onto 0 and 1

- pop():   {if (n>1) {n=n div 2;}}
- top():   {if (n==1) {return error;} else {return decode (n mod 2);}} // decode() is the inverse of code()
- size():   {return floor(log2(n));}
- empty():   {return (n==1);}

We can likewise implement a stack that stores any number (k) of symbols by using base-k numeration rather than base-2 we are used above.

Hence, for the sake of abstraction, we resolve to specify the stake by describing its externally observable behavior, without making and assumption, regardless of how vague, about its internal structure. To this effect, we specify a stack by mean of three parameters, as follows:

1) An input space, for example X, which includes all the operations that may be invoked on the stack. Hence,

X = {init, pop, top, size, empty} □ {push}□ itemtype,

Where itemtype is the data type of items we envision to store in the stack. We distinguish, in set X, between inputs that affect the state of the stack namely: AX= {init, push, pop} and inputs that merely report on it namely: VX= {top, size, empty}. 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*.

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 □ integer □ boolean □ {error}.

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

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 specifier considers correct for h. we denote this relation by *stack*, and we use the notation stack(h) 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. We present below some pairs of the form (h, y) for relation **stack**:

Stack(pop.init.push(a).init.push(b).top) =b

Stack(pop.init.push(a).size.push(b).push(c).size)=3.

Stack(init.push(a).pop.push(b).push(c).pop.empty)= false.

Stack(init.push(a).push(b).pop.pop.push(c).pop.top)= error.

Stack(init.push(a).push(b).pop.push(c).pop.pop.size)= 0.

Stack(init.push(a).pop.push(b).push(c).pop. pop.empty) = true.

We can go on describing possible input histories and corresponding outputs. In doing so, we are specifying how operations interact with each other, but we are not prescribing how each operation behaves; this leaves maximum latitude to the designer, as mandates by the principle of abstraction. It is clearly impractical to specify data types by listing elements of their relations; in the next section 4.3.6.2, we explore a closed form representation for such relations, then we choose two kinds of data types, stack and queue, whereas the remainder data types such as list, set, sequence, and multiset they explored in appendix A

4.3.10 Requirement description of some Abstract Data Types (ADTS)

4.3.8.1 Requirement description of stack

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.

- **-**operations: These are operations that alter the state of the ADT but produce no visible output.

o **Init**: this operation initializes or re-initializes the stack to empty, erasing all past history.

o **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.

4.3.8.2 Requirement description of queue

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.
  - **enq (itemtype x):** this operation adds an element (provided as parameter) at the end of queue.
  - **deq:** 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.

4.3.11 Axiomatic representation

The axiomatic specification is a formal specification defining the semantics of functions or behaviours of abstract data type by a description of its operations and the states of abstract data type on these operations, if we consider abstract data type as an object. We propose to represent the relation of a specification by means of an inductive notation, where we do induction on the structure of the input history; this notation includes two parts:

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

4.3.11.1 Specification of the stack

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

Axioms. We use axioms 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 *AX* that affect the state produce no meaningful output; hence we assume that for such input histories, the output is any element of *Y*.

Axioms:

1. Top axioms

stack(init.top) = error.

Seeking the top of empty stack returns an error.

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

Operation top returns the most recently stacked item.

2. ***Size axiom*** stack(init.size) = 0.

The size of an empty stack is zero.

3. Empty axioms

stack(init.empty)=true.

An empty stack is empty.

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

Stack that contains element *a* is not empty.

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

Let h, h' be arbitrary input histories and h+ be a non-empty input history.

1. Init rule:

stack(h.init.h') = stack(init.h').

Operation init reinitializes the state of the stack; whether it received history h prior to init or not makes no difference now (h'=()) nor in the future (h'≠ ()).

2. Init Pop rule:

stack(init.pop.h) = stack(init.h).

Pop on an empty stack has no impact now (h=()) nor in the future (h≠ ()).

3. ***Push pop rule:*** stack(init.h.push(a).pop.h+)=stack(init.h.h+).

A pop operation cancels the push that precedes it: whether we push *a* then pop it or do neither, makes no difference in the future (*h+≠ ()*). It may produce a different outcome now, hence the term *h+*.

4. ***Size rule:*** 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.

5. Empty rules

a. 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).

b. 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.

6. VX-operation rules

a. stack(init.h.top.h+)=stack(init.h.h+).
b. stack(init.h.size.h+)=stack(init.h.h+).
c. stack(init.h.empty.h+)=stack(init.h.h+).

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

We have written a closed form specification of the stack, in such a way that we describe solely the externally observable properties of the stack, without any reference to how a stack ought to be implemented; a programmer who reviews this specification has all the latitude he or she needs to implement this stack as he or she sees fit.

4.3.9.2 Specification of the queue

Axioms:

1. ***Size axiom:*** queue(init.size)= 0

The size of an empty queue is zero.

2. Empty axioms:

a. queue(init.empty)= true

An initial queue is empty.

b. queue(init.enq(a).empty)= false

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

3. Front axioms:

a. queue(init.front)= error

Invoking front on an empty queue returns an error.

b. queue(init.enq(a).enq(_)*.front)= a

Where enq(_)* designates an arbitrary number (including zero) of enq operations. Interpretation: Invoking front on a non empty queue returns the first element enqueued.

4. Rear axioms:

a. queue(init.rear)= error

Invoking rear on an empty queue returns an error.

b. queue(init.enq(_)*.enq(a).rear)= a

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

Rules:

1. ***Init rule:*** queue(h.init.h') = queue(init.h')

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

2. Init deq rule:

queue(init.deq.h) = queue(init.h)

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

3. ***Enq deq rule*** queue(init.enq(a).enq(_)*.deq.h+)=queue(init.enq(_)*.h+)

A deq operation cancels the first enq, by virtue of the FIFO policy of queues.

4. ***Size rule:*** queue(init.h.enq(a).size) =1+ queue(init.h.size) An enq operation increases the size of the queue by 1.
5. Empty rules:
   a. queue(init.h.enq(a).h'.empty) => queue(init.h.h'.empty)
   b. queue(init.h.empty) => queue(init.h.deq.empty)

Removing an enq or adding a deq to the input history of a queue makes it emptier.

6. VX-Operation rules:
   a. queue(init.h.front.h^+) = queue(init.h.h^+)
   b. queue(init.h.rear.h^+) = queue(init.h.h^+)
   c. queue(init.h.size.h^+) = queue(init.h.h^+)
   d. queue(init.h.empty.h^+) = queue(init.h.h^+)

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

4.3.12 Specification validation

In the previous section we have written specifications of the two of ADT's, namely stack and queue. How do we know that our specifications are valid, i.e. that they capture all the properties we want them to capture such as completeness and nothing else such as minimality? To bring a measure of confidence in the validity of these specifications, we envision a validation process, though by now we focus gradually on completeness; so our confidence in the validity of the specification increases. We imagine that while we are writing these specifications, an independent validation data was generated formulas of the form:

Stack(h)=y

For different values of h and y, on the grounds that whatever we write in our specification should logically imply these statements. Then the validation step consists in checking that the proposed formulas can be inferred from the axioms and rules of our specification. If they do, then we can conclude that our specification is complete with respect to the proposed formulas; if not, then we need to check with the validation data which had been generated independently to see whether our specification is incomplete, or perhaps the validation data is erroneous.

For the sake of illustration, we check whether our specification is valid with respect to the formulas written sin section 4.3.9.1 as sample pairs of input and output of our stack specification.

- ✦ V_1: $Stack(pop.\ init.\ push\ (a).\ init.\ push\ (b).\ top) = b$
- ✦ V_2: $Stack(pop.\ init.\ push\ (a).\ size.\ push\ (b).\ push\ (c).\ size) = 3$.
- ✦ V_3: $Stack(init.\ push\ (a).\ pop.\ push\ (b).\ push\ (c).\ pop.\ empty) = false.$
- ✦ V_4: $Stack(init.\ push\ (a).\ push\ (b).\ pop.\ pop.\ push\ (c).\ pop.\ top) = error.$
- ✦ V_5: $Stack(init.\ push\ (a).\ push\ (b).\ pop.\ push\ (c).\ pop.\ pop.\ size) = 0$.

For V_1, we find:

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

= {by virtue of the init Rule}

Stack(init.push(b).top) =b

= {by virtue of the push top Axiom}

b.

For V_2, we find:

Stack(pop.init.push(a).size.push(b).push(c).size)

= {by virtue of the init Rule}

Stack(init.push(a).size.push(b).push(c).size)

= {by virtue of the V-operation Rule}

Stack(init.push(a).push(b).push(c).size)

= {by virtue of the size Rule}

1+ Stack(init.push(a).push(b).size)

= {by virtue of the size Rule}

1+1+ Stack(init.push(a).size)

= {by virtue of the size Rule}

1+1+ 1+Stack(init.size)

= {by virtue of the size Axioms}

1+1+ 1+0

3.

For V_3, we find:

Stack(init.push(a).pop.push(b).push(c).pop.empty)

= {by virtue of the push pop Rule}

Stack(init.push(a).pop.push(b).empty)

= {by virtue of the push pop Rule} Stack(init.push(b).empty)= false.

= {by virtue of the push Axiom}

false.

For V_4, we find:

Stack(init.push(a).push(b).pop.pop.push(c).pop.top)= error.

= {by virtue of the push pop Rule}

Stack(init.push(a).push(b).pop.pop.top)= error.

= {by virtue of the push pop Rule}

Stack(init.push(a).pop.top)= error.

= {by virtue of the push pop Rule} Stack(init.top)= error.

= {by virtue of the top Axioms}

error.

For V_5, we find:

Stack(init.push(a).push(b).pop.push(c).pop.pop.size)

= {by virtue of the push pop Rule}

Stack(init.push(a).push(b).pop.pop.size)

= {by virtue of the push pop Rule}

Stack(init.push(a).pop.size)

= {by virtue of the push pop Rule}

Stack(init.size)= 0.

= {by virtue of init size Axioms}

0.

The appendix C includes more validation sample of ADTs**.**

4.4 SUMMARY

The chapter discussed how to ensure the validity of specification by introduced relational mathematics. In relational mathematics we represent specifications in sets and relations notation. The concepts of sets and relations are used to ensure that the completeness and minimality of specifications are valid this chapter explain the following relational mathematics concepts: First of all, algebra of relations, including operations, and properties. Second, principles of sound specification, and how relation support theses. Third, it expressed the concepts of the join of relations, its significance, and its role in specification validation. Fourth, it explained, the relational specification of systems that maintain an internal state. Fifth, it expressed the axiomatic representation of the relational specification of systems that maintain a state. Last, it discussed the generation and validation of axiomatic specifications.

CHAPTER FIVE

ALNEELAIN SPECIFICATION LANGUAGE

5.1 INTRODUCTION

This chapter present Alneelain Specification Language NSL that we are proposed to use to specify abstract data types. We used BNF notation to construct the structure of the language. Alneelain Specification Language is built around the proposed specification model that presented in chapter one.

Alneelain Specification Language tokens are illustrated. The description of the stack abstract data type and its specification using Alneelain specification language are shown in this chapter.

5.2 ALNEELAIN SPECIFICATION LANGUAGE NSL

Specification languages are looked at as artificial languages defined by scholars initially for the purpose of communicating with computers but, as importantly, for communicating algorithms among people [41] Unlike a program, in formal specification specifications written in a specification language is not necessarily planned to be run on the target platform. The use of a specification language makes analysing and simulating alternative system solutions possible task. Particularly, a specification language provides a user of a language with a well-defined set of concepts, thereby improves his/her capability for both working out a solution to a problem and reasoning about the solution [42]. Our steps toward building Alneelain Specification Language NSL, firstly we use BNF to describe the language. BNF is a Meta language which used to describe a programming language [43]. BNF named refer to John W. Backus and Peter Naur. Using BNF it is possible to specify which the sequences of the symbol constitute a syntactically valid program in a given language. The semantics issue that is expressed what such valid of symbols mean must be specified separately. A discussion of the Alneelain Specification Language BNF is follows.

5.3 ALNEELAIN SPECIFICATION LANGUAGE BNF

Like proposed by Peter Backus Naur in Algol 60 [44], we proposed Alneelain Specification Language BNF approximate to Algol 60. The expression or the formula defined terms whose names are enclosed in angle brackets. For example <abcd>. Each of these denotes a term or an expression of basic language symbols. The metalanguage of Alneelain Specification Language BNF is supposed to describes the expression or formula of our language. An expression or formula is sequences of characters enclosed in brackets represent metalanguage terms whose values sequences of symbols. The mark " ::= " is metalanguage connective separate left-hand side and right-hand-side, the "|" is metalanguage connective mean "or". Any mark in a formula, which is not term or connective, denotes itself. And words are not in the brackets are keyword that have special meaning in the language such as specification, constant, type, input, endinput, …,. Those keywords are appearing in bold style. Let us show this example:

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

In this example Alneelain is an expression that can be comprises the words in the left hand side after the symbols::= header, body, and the keyword endspecification.

There are some symbols have meaning in our expression, for example the colon ":" separates between identifier and its type, the "=>" and "=" are separate left hand side from the right hand side of axioms and rules, the coma "," is separates between two or more than terms in the right side of the formula, the "^" is used to concatenates two or more that two characters in one set enclosed in braces and they separated by coma ",". The following section it presents the BNF

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

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

<specname>::= < identifier>

< identifier>::=<letter>$^{+}$ \\ **+** ***means repeating one or more times***
<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>] ; \\ **[ ]** ***means optional item***

[<typesection>;]

<inputsection>;

<outputsection>;

<varsection>;

<axiomsection>;

<rulesection>;

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

<constbody>::=<constant> | <constant> , < constbody>

<constant>::=<constname> = <constvalue>

<constname>::=< identifier>

<constvalue>::= <digit>$^{+}$

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

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

<typebody>::= <typedeclaration>

| <typedeclaration> , < typebody>

<typedeclaration>::=< identifier> : <typedefinition>

<typedefinition>::=< identifier>

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

<voppart> ::= <vopdeclaration>

| <vopdeclaration> , <voppart>

<vopdeclaration>::= **vop**<methodname>[**(**<paramtype1>,<paramtype2>**)**] **: <**returntype>

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

<oopdeclarations> ::= <oopdeclaration>

| <oopdeclaration> , <oopdeclarations>

<oopdeclaration> ::= <methodname> [**(**<paramtype1>,<paramtype2>**)**]

<methodname>::= <identifier>

<paramtype>::=<identifier>

<returntype>::=<identifier>

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

<outputlist>::=<outputtype>

| <outputtype>

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

<outputtype>::= < identifier>

<varsection> ::= **var** <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> ::= <specterm> <operators> <specterm>

<operators> ::=   => | =

5.4 ALNEELAIN SPECIFICATION LANGAUGE TOKENS

As a token is defined of basic component of the source code of the specific language, we present all the tokens of our language which are classified in eight types of tokens that can be used to describe their formulas and expressions. These expressions are including constants, identifiers, operators, reserved words, digits, special characters, white space, newline, and tab space. The following table 5.1 illustrates all tokens of Alneelain Specification Language

Table 5.1: Alneelain Specification Language tokens.

Token type	Token
Keywords	specification, endspecification, type, constant, input, endinput, vop, oop, output, endoutput, variable, axioms, axiom, endaxioms, rules, endrules, rule
Digits	0 – 9
Identifiers	[a-z A-Z] [a-z A-Z]*
Operators	= , => , + , - , > , < , >= , <=
Special characters	**{** , **}** , **(** , **)** , **;** , **:** , ^ , **&** , **,**
Whitespace	‘ ‘
Newline	\n
Tab	\t

5.5 REQUIREMENT OF ABSTRACT DATA TYPE TOWARD USING ALNEELAIN SPECIFICATION LANGAUGE

In this section we present a sample of abstract data type from requirement description to specification of abstract data type from the view of the researcher to specification using Alneelain Specification Language. The following example illustrates a set abstract data type. Which the set is an abstract data type that can store certain values, without particular order. In addition, a set has no duplicate element. The important thing that we must be sure that there are no duplicates.

5.5.1 Requirement description of a stack

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. The following description of a stack contain both O-operation and V-operation

<u>O</u> -operations: These are operations that alter the state of the ADT but produce no visible output.

<u>init</u>: this operation initializes or re-initializes the stack to empty, erasing all past history.

<u>push (itemtype x):</u> this operation pushes an element (provided as parameter) on top of the stack.

<u>pop:</u> this operation removes the top (most recently pushed) element of the stack, if the stack is not empty; else it leaves the stack unchanged.

<u>V</u>-operations: These are operations that return values but do not change the state.

<u>itemtype: top():</u> returns the top of the stack (last element stored) if the stack is not empty; else it returns an error message. **<u>integer: size()</u>**: returns the number of elements of the stack.

<u>boolean: empty()</u>: returns true if and only if the stack is empty.

5.5.2 Specification of a stack using Alneelain Specification Language

Here we use Alneelain Specification Language NSL to specify the stack; the following statements illustrate

```
 specification Stack;
constant
        x = 7;
type
        itemtype : char;
input
        vop top(): itemtype ,
        vop size: integer ,
        vop empty: boolean
        oop init, pop(), push()
endinput;
output
        char  ^  Boolean  ^  integer ^  error endoutput; variable  a: char ,  n: char,  h:
undefined,
        hprime: inputstar ,
        hplus: undefined;
axioms
         axiom topAxiom:
                Stack(init.top) = error &
                Stack(init.h.push(a).top) = a,
axiom sizeAxiom:
```

```
            Stack(init.size) = 0,
axiom emptyAxiom:
            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) = Stack(init.h.size) ,      rule emptyRule:
            Stack(init.h.push(a).hprime.empty)=>            Stack(init.h.hprime.empty)&
Stack(init.h.empty) => Stack(init.h.pop.empty) ,      rule vopRule:
            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
```

Appendix B. Illustrate the specifications of some of abstract data types using Alneelain Specification language for the *queue, sequence, set, multiset, and list respectively*.

5.5.3 Specification of the stack in object-Z

In order to specify the bounded stack, we defined a constant say *max,* beyond which the size of stack cannot grow. The following figures 5.1 show the stack specification using Z specification language

$$\begin{array}{|l}
\textit{INIT} \\ \hline
items : \mathrm{seq}\ T \\ \hline
\# items \leqslant max \\
items = \langle\, \rangle \\ \hline
\end{array}$$

Figure 5.1: initiate stack in Z

Figure 5.1 shows a possible initialization state of the stack using schema. It gives the initial configuration of the system

$$\begin{array}{|l}
\textit{Push} \\ \hline
\Delta(items) \\
items, items' : \mathrm{seq}\ T \\
item? : T \\ \hline
\# items \leqslant max \\
\# items' \leqslant max \\
\# items < max \\
items' = \langle item? \rangle \frown items \\ \hline
\end{array}$$

Figure 5.2: push operation of stack in Z.

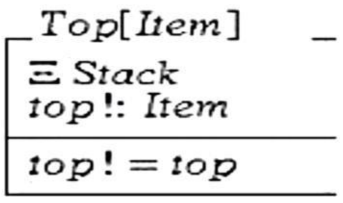
Figure 5.2 shows push operation of the stack, printing (′) of variable denotes the after state that component thus *item′* refer to the after state of the variable items.

An operation Δ-list contains a subset of the variables which are declared.

$$\begin{array}{|l}
\textit{Pop} \\ \hline
\Delta(items) \\
items, items' : \mathrm{seq}\ T \\
item! : T \\ \hline
\# items \leqslant max \\
\# items' \leqslant max \\
items \neq \langle\, \rangle \\
items = \langle item! \rangle \frown items' \\ \hline
\end{array}$$

Figure 5.3: pop operation of stack in Z.

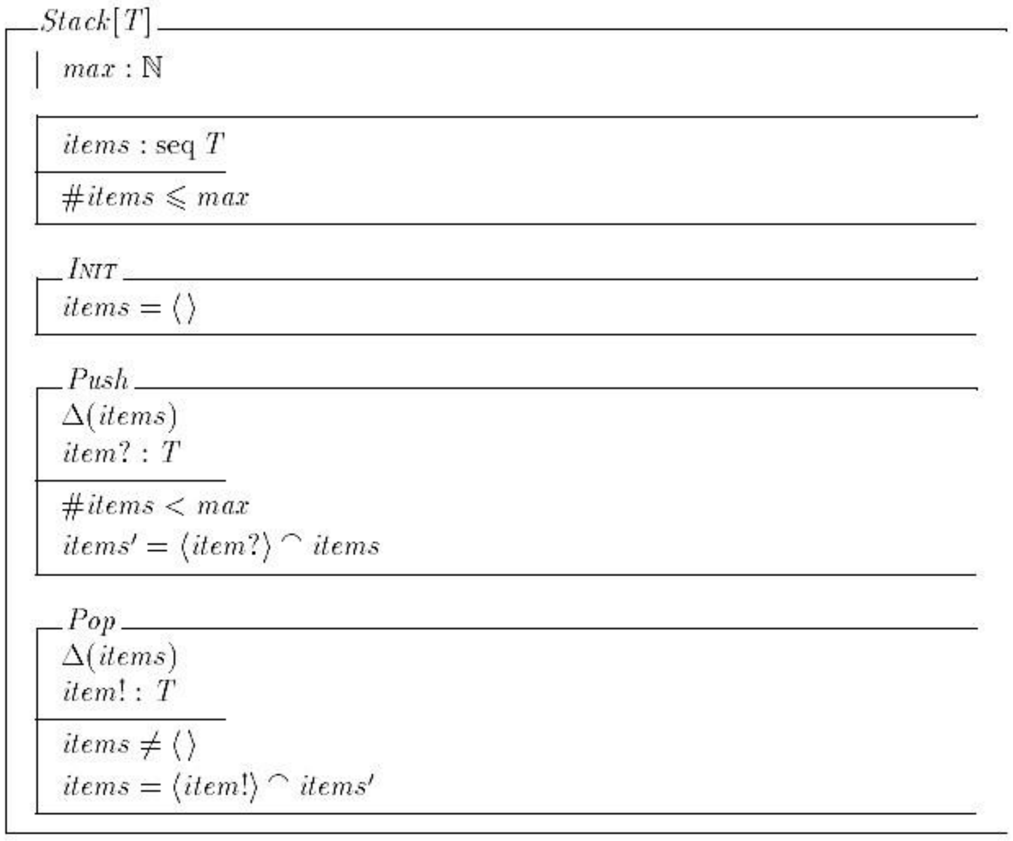
Figure 5.3 shows pop operation of the stack,



```
Top[Item]
Ξ Stack
top!: Item
─────────
top! = top
```

Figure 5.4: top operation of stack in Z.

Figure 5.4 shows top operation in the stack,



```
Stack[T]
│ max : ℕ

  items : seq T
  ─────────
  #items ⩽ max

  INIT
  items = ⟨⟩

  Push
  Δ(items)
  item? : T
  ─────────
  #items < max
  items' = ⟨item?⟩ ⁀ items

  Pop
  Δ(items)
  item! : T
  ─────────
  items ≠ ⟨⟩
  items = ⟨item!⟩ ⁀ items'
```

Figure 5.5: a full stack specification in Z.

Figure 5.5 shows three stack operations which are initialization, push, and pop and explains the state of the stack in each case and the constraints that must be considered.

5.6 SUMMARY

In this chapter we have used the concepts of Backus-Naur Form BNF that are used as metalanguage to describe Alneelain Specification Language, the BNF of our language , requirements description of stack as an example of abstract data type requirements, and specification of a stack using our Alneelain Specification Language, specification of a bounded stack in Z specification language.

CHAPTER SIX

ALNEELAINCOMPILER

6.1 INTRODUCTION

In this chapter we introduce Alneelain compiler which is composes of two components of the language; they are lexical analyzer and syntax checker. The lexical analyzer uses to check that all the tokens of such specified abstract data types are true whereas the syntax checks the specification of abstract data types is correct

6.2 COMPLIER DESIGN

A compiler is a program that takes the input of the specific language written in a high-level and produces the output. Any a specification language needs to be complemented with the construction of its complier. This will enable the language to be useful and formal. To this we must have two steps process. In the first step; the source program will be compiled and in the second step is loaded into memory and then executed. It generally uses a front end and a back end [45]. In our work case the compiler does not need to includes all compiler components such as pre-processing, lexical analysis (scanning), syntax analysis (parsing), combining scanning and parsing, semantic analysis, intermediate code generation until the target code as shown in figure 6.1. It need only two first components as shown in methodology in figure 1.1 both of them regard as a front end type which are lexical analyser and syntax checker; lexical analyzer is used to convert source code of specification language into a sequences of tokens, characters, strings, terminated by spaces [46] In the following two sections we introduce more about lexical analyser and syntax checker.

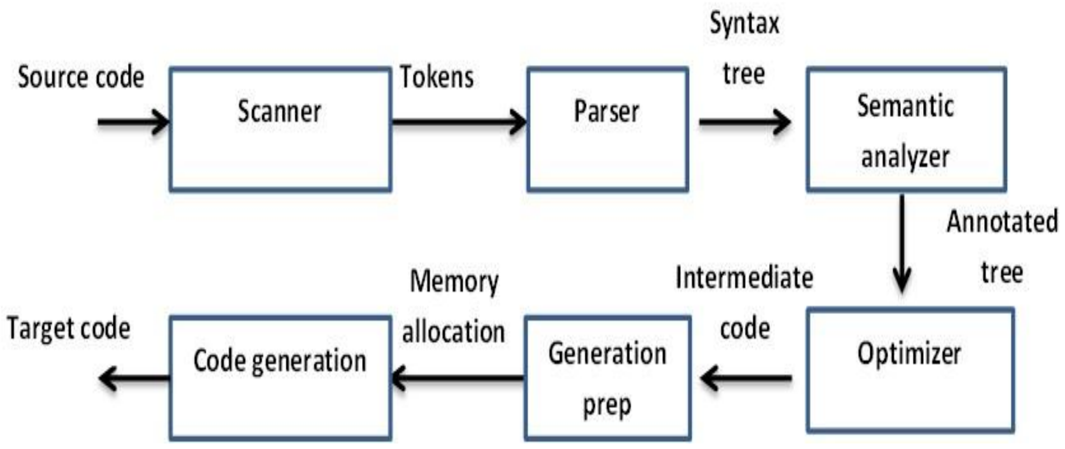


Figure 6.1: Compiler components from cs.uni.edu.

6.2.1 Lexical analyzer

Lexical analysis is the process of analyzing a stream of individual characters into a sequence of lexical tokens to feed into the parser [47], which is called tokenization. Tokenization is means instance of words and punctuation symbols that make up source code [47]. A program that performs lexical analysis may be called a lexer or scanner [48]. Such a lexer is generally combined with a parser, which together is analyzing the syntax of the specification language [49]

6.2.2 The tokens

A token is categorizing block of text, usually consisting of indivisible characters known as lexemes a lexical analyser initially reads in lexemes and categorizes them according to function, giving them meaning [47]. This assignment of meaning is known as tokenization. A token can look like anything: English words, symbols, digits, anything; it just needs to be a useful part of the structured text [47]. A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyser as an instance of that token [45].

6.2.3 Lexical analyzer of Alneelain Specification Language NSL

The following statements express our lexical analyser:

```
  void gettoken()

{

          switch (c) {

case '.':  token =dot ; nextchar(); break; case ',':   token=comma; nextchar(); break; case
';':   token=semicolon; nextchar(); break;

          case ':':   token=colon; nextchar(); break;            case '^':   token=unionsign;
nextchar(); break;            case '(':   token=lparen; nextchar(); break;            case ')':
token=rparen; nextchar(); break;            case '{':   token=lbraces; nextchar(); break;
case '}':   token=rbraces; nextchar(); break;            case '+':   token=plussign;
nextchar();break;            case '-':   token=minussign; nextchar();break;            case '>':
token=greatersign; nextchar();      if (c=='=')

            {                         token=greaterequalsign; nextchar();

                              } break;

 case '<':  token=lesssign; nextchar();            if (c=='=')     {

   token=lessequalsign; nextchar();

                 } break;

      case '&':  token=Ampersand; nextchar(); break;        case ' ':  nextchar();
gettoken(); break;        case '\n': nextchar(); gettoken(); break;        case '\t': nextchar();
gettoken(); break;        case '=':  token=equalsign; nextchar();               if (c=='>')

{

                              token=impliessign; nextchar();
```

```
                } break;        default:

          if (isalpha(c))               {               while (isalpha(c))

                 {          temp[i] = c;          nextchar();          i++;
}          temp[i]='\0';          tokcontent =temp;          i=0;
if(tokcontent=="specification") {token=specification; goto end;}
if(tokcontent=="endspecification") {token=endspecification; goto end;}
if(tokcontent=="type") {token=type; goto end;}   if(tokcontent=="constant")
{token=constant; goto end;}       if(tokcontent=="input")  {token=input; goto end;}
if(tokcontent=="endinput")    {token=endinput; goto end;}
if(tokcontent=="vop"){token=vop; goto end;}
if(tokcontent=="oop"){token=oop; goto end;}          if(tokcontent=="output"))
{token=output; goto end;}          if(tokcontent=="endoutput"){token=endoutput;
goto   end;}   if(tokconten==variable"){token=variable; goto end;}
if(tokcontent=="axioms"){token=axioms; goto end;}  if(tokcontent=="endaxioms")
{token=endaxioms; goto end;}      if ((tokcontent=="axiom") {token=axiom; goto
end;}        if(tokcontent=="rules"){token=rules; goto end;}  if(tokcontent=="endrules")
{token=endrules; goto end;}      if(tokcontent=="rule"){token=rule; goto end;}
if(1)token=identifier;       goto end;     }

          if(isdigit(c))              {            while (isdigit(c))

                {                         temp[i] = c;                    nextchar();

                         i++;     }            temp[i]='\0';
constvalue=temp ;           i=0;                token=digit;              goto
end;            }           end:

                    break;    }

               }
```

6.2.4 Syntax checker

When an input string (source code of the language) is given to a complier, the compiler processes it in several phases, starting with the previous lexical analysis phase which scans the input and divide it into tokens [47]. Syntax analysis or parsing is the second phase, i.e. is after lexical analysis. It checks the syntactical structure of the given input, i.e. whether the given input is in correct syntax of the language in which the input has been written or not. It does so by building a data structure, called a parse tree or syntax tree. The parse tree is constructed by using the pre-defined Grammar of the language and input string [47]. Syntax error can be detected at this level if the input is not in accordance with the grammar. If the given input string can be produced with the help of the syntax tree, the input string is found to be in the correct syntax.

6.2.5 Syntax analyzer of Alneelain Specification Language

It is a quite simple tool, it can be used to check and find syntax errors of our Alneelain Specification Language files recursively. The following code statements express our lexical analyser

```
 void Alneelain( )    {//   <Alneelain> ::=  <header> ; <body> endspecification
diagnosis=true;     header( );     checktoken(semicolon);     body( );
checktoken(endspecification);     if (diagnosis) {           cout<<"syntactically correct"; }
else {           cout<<"syntactically incorrect"; }} void header( )    { //  <header> ::=
specification <specname>      checktoken(specification);      specname = tokcontent;
gettoken();} void body( ) {//  <body> ::=  [<constsection>;]
[<typesection>;]<inputsection>;

    <outputsection>;  <varsection>; <axiomsection>; <rulesection>     if
(token==constant) {constsection( );  checktoken(semicolon);}     if (token==nlntype)
{typesection( );  checktoken(semicolon);}     inputsection( );  checktoken(semicolon);
outputsection( );  checktoken(semicolon);

    varsection( );  checktoken(semicolon);     axiomsection( );  checktoken(semicolon);
rulesection( ); checktoken(semicolon);}.
```

The above statements show the logical expression to perform the syntax analyser. Each of section has more functions within it to assist the main section module to perform its code for example, the input section has two functions they are: voppart(), and ooppart(), output section has outputlist() function, axiom section has two functions they are: axiomslist(), and axiombodies, and the rule section has two functios they are: rulelist(), and rulebodies functions.

ALNEELAIN SPECIFICATION LANGUAGE USER INTERFACE

Figure 6.2: shows Alneelain Specification Language NSL interface. The main window is appearing after select Alneelain Specification Language NSL item from the program menu. It allows user to create a new file that contains a specification of one abstract data type or any other specification that can be written in the syntax of Alneelain Specification Language NSL. Also, it allows user to check if the specification is correct and meet the syntax of the language or not. It has sufficient documentation that the users can read it before using the language. Alneelain Specification Language contains four basic menus and one toolbar. The toolbar provides instant access to the most frequently use commands in the menus. The four menus are explained as follows:

2- File Menu:

By opening the file menu a user can do the following:

- **New file**: this item prompts user to create a file with an empty text editor. A user can write the specification in the syntax of Alneelain Specification Language using keyboard. The cursor will be appearing in text editor.
- **Open file**: this item prompts user to open an existing file by determined the name and location of the file, and then the file will opened in the application.
- **Save file**: this item allows user to save the file that is open in the current tab of the editor to the current selected name with an extension ".nln".

- **Save As file**: this item allows user to save the file that is open in the current tab of the editor to the different name with an extension “.nln”.
- **Exit**: this item quits the application. It prompts the user to save any changes occurred to the file that was opened in the editor.

3- Edit Menu:

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

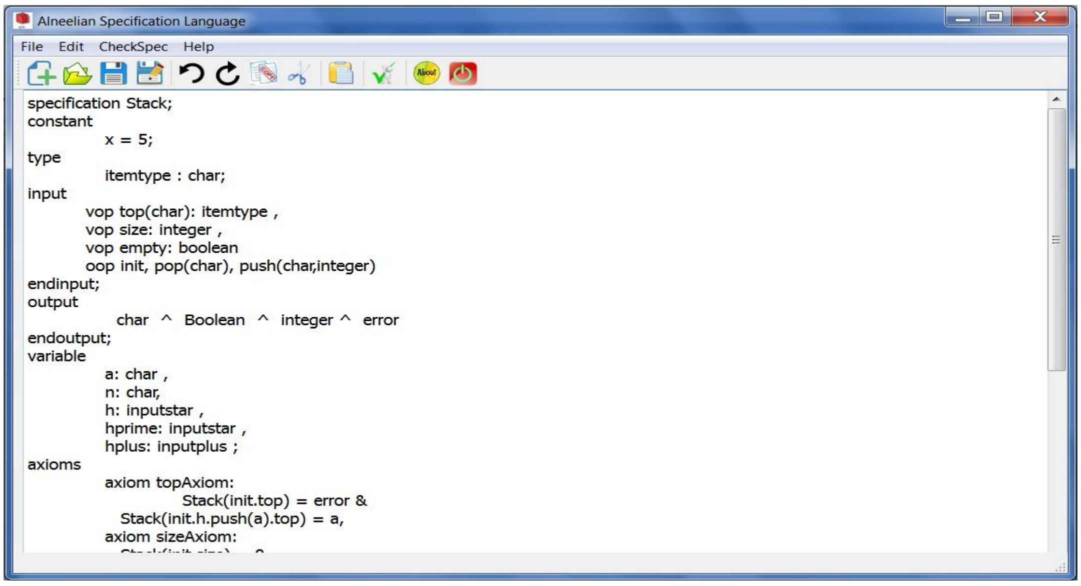
4- CheckSpec Menu:

This menu causes the opened file to be processed. It can use to checks file for syntax of the language. After performing a check, a confirmation message appears confirming that the specification is syntactically correct or not. Figure 6-3 shows that our application checking the specification of one of abstract data type named stack.

5- Help Menu:

This menu provides help on using the Alneelain Specification Language software

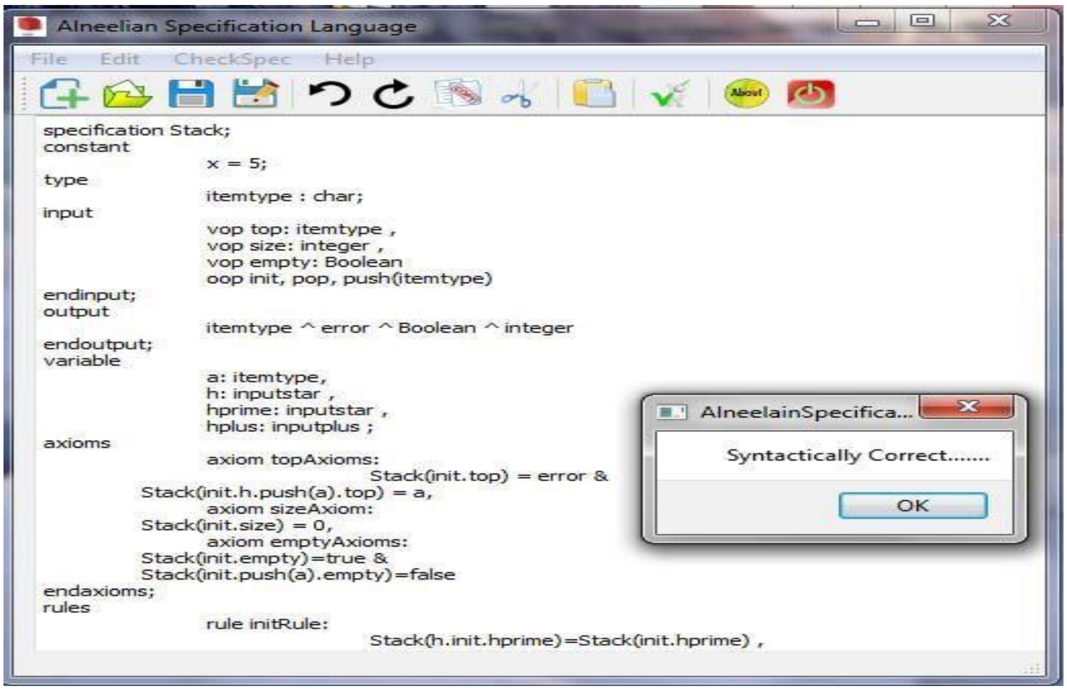
The following figure 6.2 illustrates Alneelain Specification Language NSL:



```
specification Stack;
constant
            x = 5;
type
            itemtype : char;
input
         vop top(char): itemtype ,
         vop size: integer ,
         vop empty: boolean
         oop init, pop(char), push(char,integer)
endinput;
output
              char  ^  Boolean  ^  integer ^  error
endoutput;
variable
            a: char ,
            n: char,
            h: inputstar ,
            hprime: inputstar ,
            hplus: inputplus ;
axioms
            axiom topAxiom:
                        Stack(init.top) = error &
              Stack(init.h.push(a).top) = a,
            axiom sizeAxiom:
```

Figure 6.2: Alneelain Specification Language NSL - Main Window.

This is the main window application of Alneelain Specification Language NSL.



```
specification Stack;
constant
                    x = 5;
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) ,
```

```
Syntactically Correct.......
```

Figure 6.3: Checking of the Stack.

The figure 6.3 above illustrates the checking of stack that specified using Alneelain Specification Language.

6.2.6 The output of Alneelain Specification Language NSL

After checked that an abstract data type was syntactically correct an output has been produced. This output shows all axioms and rules for that abstract data type, for example after checking a stack as shown in figure 6.3 and stack specification as shown in section 5.5.2 is as follows:

```
topAxioms:

                              Stack(init.top) = error &

             Stack(init.h.push(a).top) = a,

sizeAxiom:

             Stack(init.size) = 0,

emptyAxioms:

             Stack(init.empty)=true &

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

\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

initRule:

                    Stack(h.init.hprime)=Stack(init.hprime) , initpopRule:

                    Stack(init.pop.h) = Stack(init.h) , pushpopRule:

                    Stack(init.h.push(a).pop.hplus) = Stack(init.h.hplus) , sizeRule:

                    Stack(init.h.push(a).size) = 1 + Stack(init.h.size) ,

 emptyRules:

                    Stack(init.h.push(a).hprime.empty)=> Stack(init.h.hprime.empty)&
```

Stack(init.h.empty) => Stack(init.h.pop.empty) ,

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)

6.3 TOOL USED FOR DEVELOPING

We used Qt creator to develop our tool because our base on C++ programming language, in addition Qt Creator is across platform for both C++ and JavaScript and has OML integrated development environment which is part of the SDK for Qt GUI application, Qt Creator uses the C++ compiler from the GNU compiler on Linux and FreeBSD. And then uses MinGW or MSVC with the default install. That is suitable for developing.

6.4 SUMMARY

In this chapter we have introduced the concepts of complier, compiler components, compiler design, two major components that we need in our language they are lexical analyser and syntax checker, the concepts of tokens and lexeme, Alneelain Specification Language NSL user interface and does a user can use it, and C++ programming language under Qt 5.1.1 as tool used to develop our language.

CHAPTER SEVEN

ALNEELAINVALIDATION TOOL

7.1 INTRODUCTION

In this chapter the steps that we followed toward create AlneelainValidation Tool (NVT) will introduced, which take the abstract data types that had been outputted from Alneelain compiler to become an input to the validation tool. The results of abstract data types that obtained after checked their syntax includes axioms and rules in a log file. Rewriting algorithm was implemented to perform this. A user can write independent validation data and then use the tool check whether these validation data were valid or not

7.2 TERM REWRITING

Term rewriting is a surprisingly simple computational paradigm that is based on the repeated application of simplification rules [50], [51]. It is particularly suited for tasks like symbolic computation, program analysis and program transformation [51]. In our work term rewrite is use to simplify queries to specific output. Recall the example given in section 4.3.10

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

by virtue of the init rule, which is in form; Stack(h.init.hprime)=Stack(init.hprime) supposed that *pop,init.push(a)= "h"*, and *push(b).top = hprime*, and stores hprime, we found that Stack(init.hprime), after restore hprime we found:

Stack(init.push(b).top) =b

After search in all the rule we cannot found any rule match the form Stack(init.push(b).top), that means we can check axioms then we found the push top axiom match the form. If the matching make the left-hand-side and the right-handside are identical, we write the right side of the axiom as a result, otherwise, an error was occurred. Figure 7-1: shows rewriting algorithm.

7.3 REWRITING ALGORITHM

The algorithm has some aspects to be specified before proceeding to implement it; there are two important strategies to be select; first, in what way is the redex selected? Second, in what order are the rules can be applied? There are many possible methods for selecting the redex which are the following:

- From the root of the string to the leaves ▸ From the leaves to the root.
- From left to the right.
- From right to the left.

But we chose from the left to the right because it is closer and more convenient to apply to our queries. Also there are various methods exist for selecting the rules to be applied which are the following:

- Textual order.
- Specificity order (rules with more precise left-hand sides are tried before rules with more general left-hand sides).

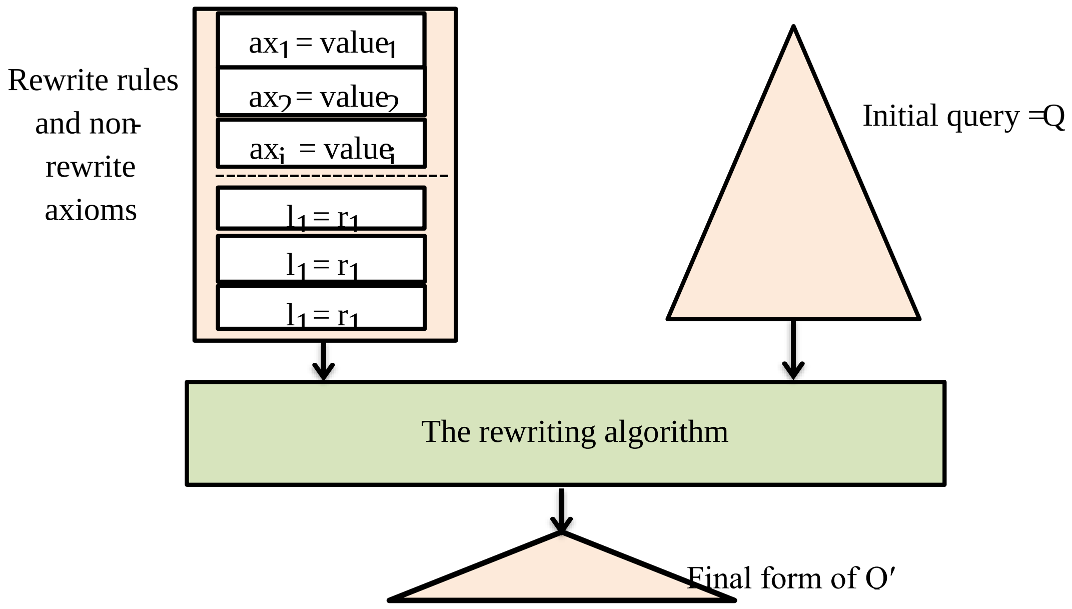


Figure 7.1: Rewriting Algorithm source: http://www.meta-environment.org, Paul.

In our work we used mixed technique no order method but some time used textual order due to flexibility. Figure 7.2 shows more details for the rewriting algorithm.

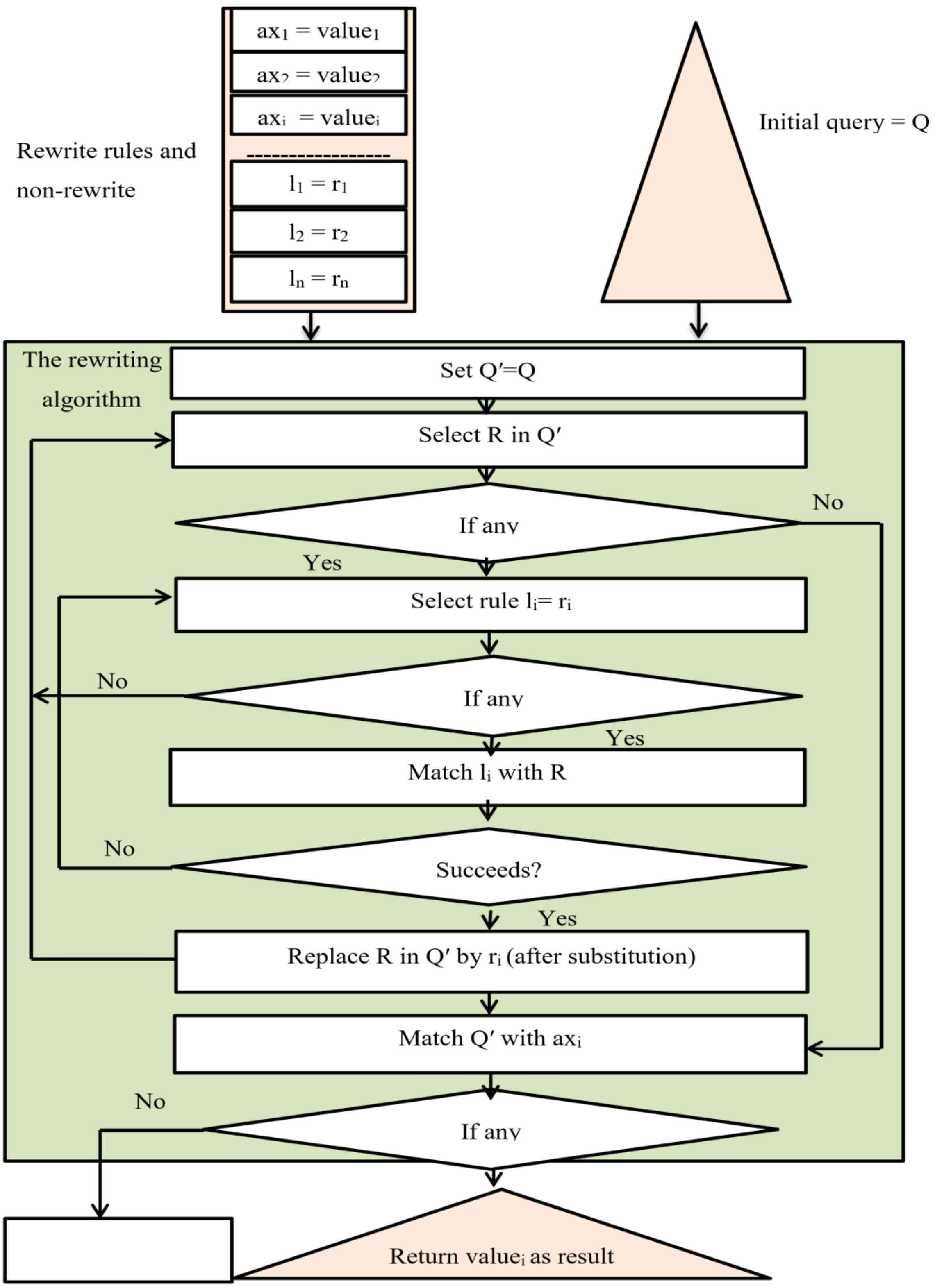


Figure 7-2: rewriting algorithm details

Figure 7.2: Rewriting Algorithm more details.

7.4 ALNEELAINVALIDATION TOOL NVT

The main idea of the AlneelainValidation Tool NVT is that the tool has two types of handle the query entered by a user takes a formula from a user of the form:

- *Stack(push(a).init.pop.push(a).top.pop.size)* then simplifies it until it finds 0, or in as follows:
- Stack(init.push(b).push(g).pop.push(k).pop.push(c).empty), and simplifies it until finds false.

Or it takes a formula from a user of the form:

- *stack(push(a).init.pop.push(a).top.pop.size)=0*, and proves that it equivalent to false. or in as follows:
- Stack(init.push(b).push(g).pop.push(k).pop.push(c).empty)=false, and proves that it equivalent to true

To perform this we suppose that all axioms and rule that had been check by syntax analyser of Alneelain Specification Language NVT are hypotheses that the algorithm worked in.

7.4.1 Simplification method

Simplification of any query, we suppose that an initial expression let Q must be simplified in a number of times in a number of rules. The initial expression Q may have complex left-hand side that can be simplified into non-simplified expression. There are many rules also have complex left-hand side that simplified into the expression appearing in the right-hand side. The initial Q may gradually reduce.

When the initial expression can be reduced we called reducible expression *redex*

[50]. If the initial expression can't be reduced that means we reach the final result by comparing the expression with axiom. The following steps explain the idea:

1. An initial expression Q that is to be simplified
2. Finding a match two cases
    a. If there is a match between the *redex* and the left-hand side of the axiom, print out the right-hand side of that axiom as a result.
    b. If there is a match between the *redex* and the left-hand side of the rule, reduces an expression to be same as the right-hand side of that rules.
3. Repeat b until there is no matching between the *redex* with any rule.
4. If there is no a match between the *redex* and any rule that means there must be a match between the *redex* and left-hand side of one axiom. In this case mean that we reach the final matching and print out the right-hand side of the match axiom as a result.

7.5 ALNEELAINVALIDATION TOOL USER INTERFACE

Figure 7-3: shows the main window of AlneelainValidation Tool NVT. This main window appears when a user selects it from the program menu. It allows user to write his query and check it. There are two types of checking a user query:

1. A user write query of form *Stack(push(a).init.pop.push(a).top.pop.size)* and then click first Run command button, the tool return only the final result equal to 0.
2. A user write query of form *Stack(push(a).init.pop.push(a).top.pop.size)=0?* Without spaces between "), =, 0, and?", and then click second Run command button, the tool return true.

Notice that the type one act as rewrite system and the second one act as theorem prover system. The master type is the type one, so the type two act as supporting to type one. It also allows user to determine the operation name for example he can set *start* instead of *init* and so on.

AlneelainValidation Tool NVT contains two basic tabs:

1. Check tab:

This tab has comobox that enable user to select data type from it, it has two texts box that enable user to write their queries, first text box for rewriting query whereas the second one for theorem proving queries. Each text box has command button one the left side of text enable user to click them to show their results. Also, each text box has command button one the right side of text enable user to click to erase the text box or to delete the query from it. There are two commands to delete the result.

2. Settings tab:

This tab has many text boxes enable user to change abstract data type operation.

The user can type different operations for example:

The user cans changes *init* and select *start* instead of it.

The user cans changes *push* and select *add* instead of it.

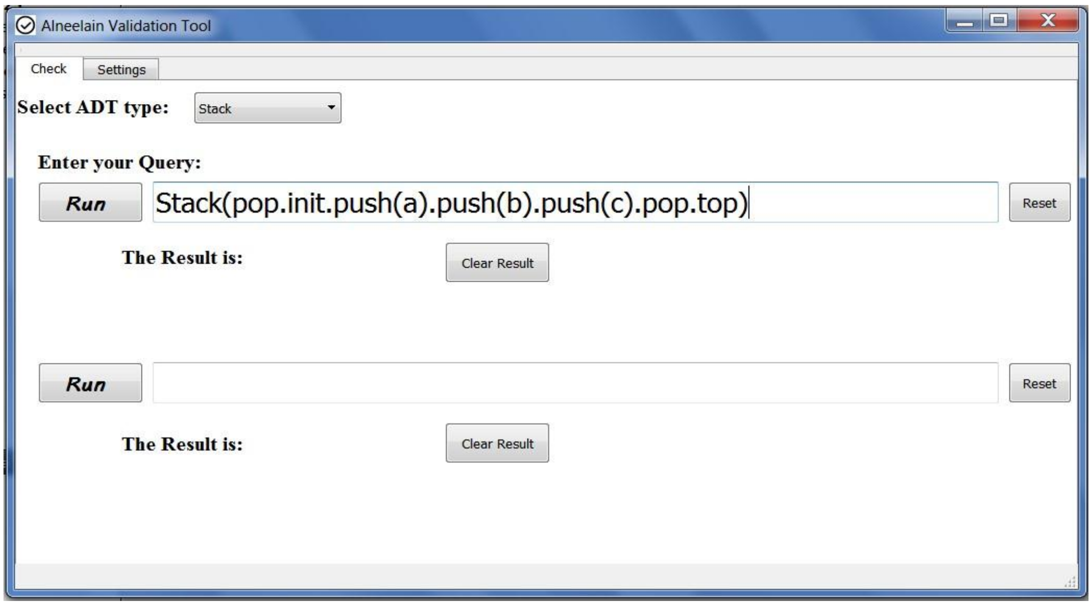
The user cans changes *pop* and select *del* instead of it.

The user cans changes *enq* and select *addq* instead of it.

The user cans changes *deq* and select *deleteq* instead of it.

The user cans changes *rear* and select *tail* instead of it.

These changes above are just examples but a user can select any operations for any abstract data types. Notice that a user he wants to change these operations must set and select operations before him writing his queries on the check tab. After changing operation a user must click on the set command button on the right bottom of the window.



Select ADT type: Stack

Enter your Query:

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

The Result is:

The Result is:

Figure 7.3: AlneelainValidation Tool NVT- Main window.

7.6 TOOL USED

We used Qt creator to develop our tool because our base on C++ programming language, in addition Qt Creator is across platform for both C++ and JavaScript and has OML integrated development environment which is part of the SDK for Qt GUI application, Qt Creator uses the C++ compiler from the GNU compiler on Linux and FreeBSD. And then uses MinGW or MSVC with the default install. That has been suitable for developed AlneelainValidation Tool.

7.7 SUMMARY

In this chapter we have presented the concepts of term rewriting and its applications area in software engineering, rewriting algorithm that applied to perform simplification as important technique toward validation, ac clear flow chart of rewriting algorithm was also presented , the simplification method that the rewriting algorithm use to perform their work, a user interface of AlneelainValidation Tool NVT was presented with explaining more detail on how does it work and how a user can use it, and finally, the chapter introduced the software tool that used to build AlneelainValidation Tool.

CHAPTER EIGHT

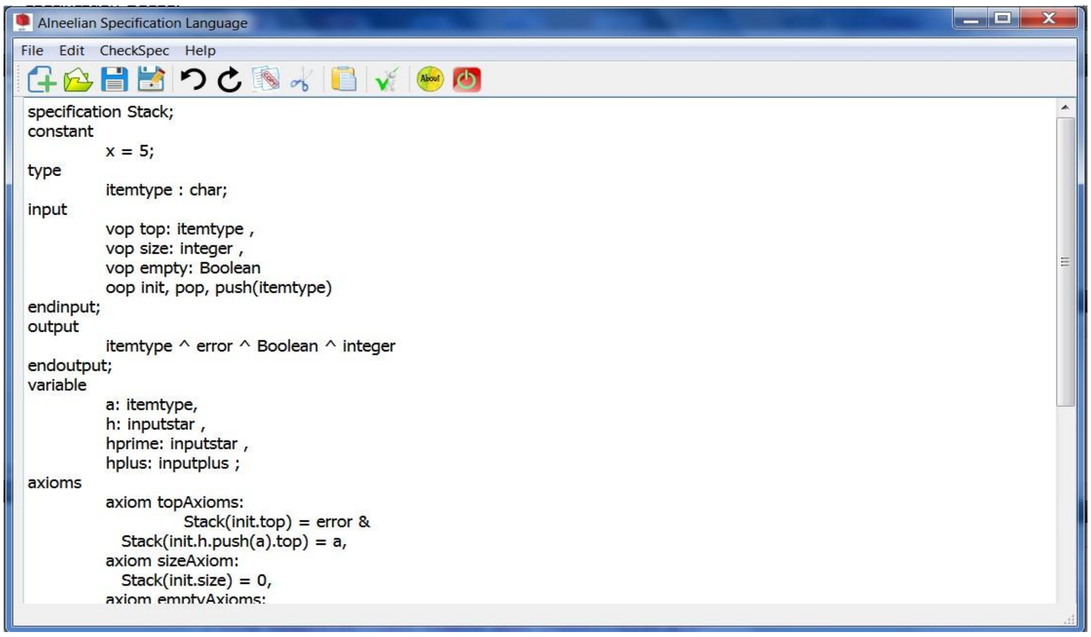
DEPLOYMENT AND ILLUSTRATION

8.1 INTRODUCTION

This chapter we'll provide an illustration of Alneelain Specification Language NSL and AlneelainValidation Tool NVT in more details represented in windows and gives an interpretation for each window and explains how a user should do to check their queries.

8.2 AN ILLUSTRATIONS OF ALNEELAIN SPECIFICATION LANGUAGE

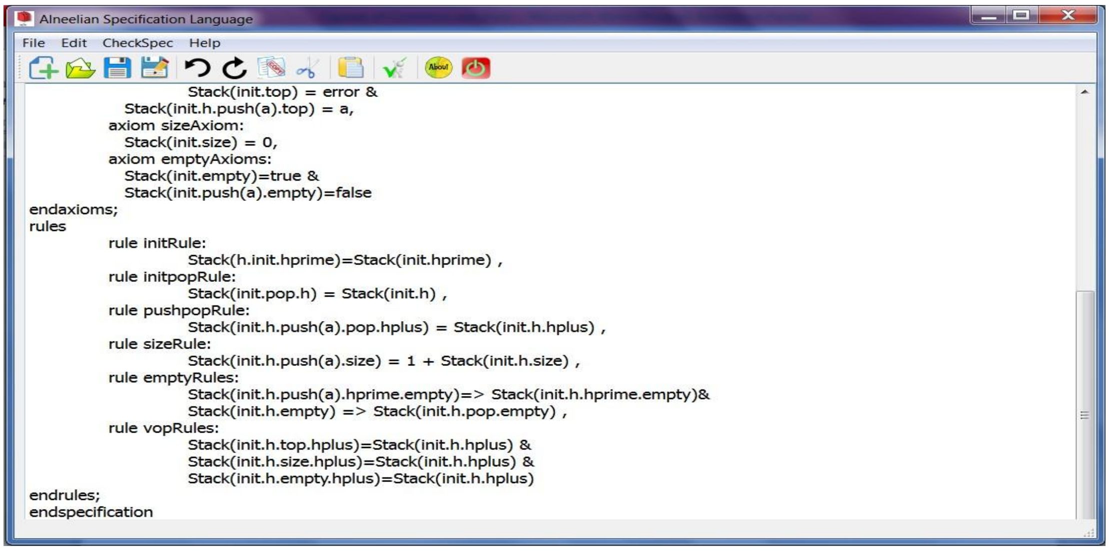
Here are some windows represent Alneelain Specification Language NSL



```
specification Stack;
constant
          x = 5;
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:
```

Figure 8.1: Alneelain Specification Language: Main Window.

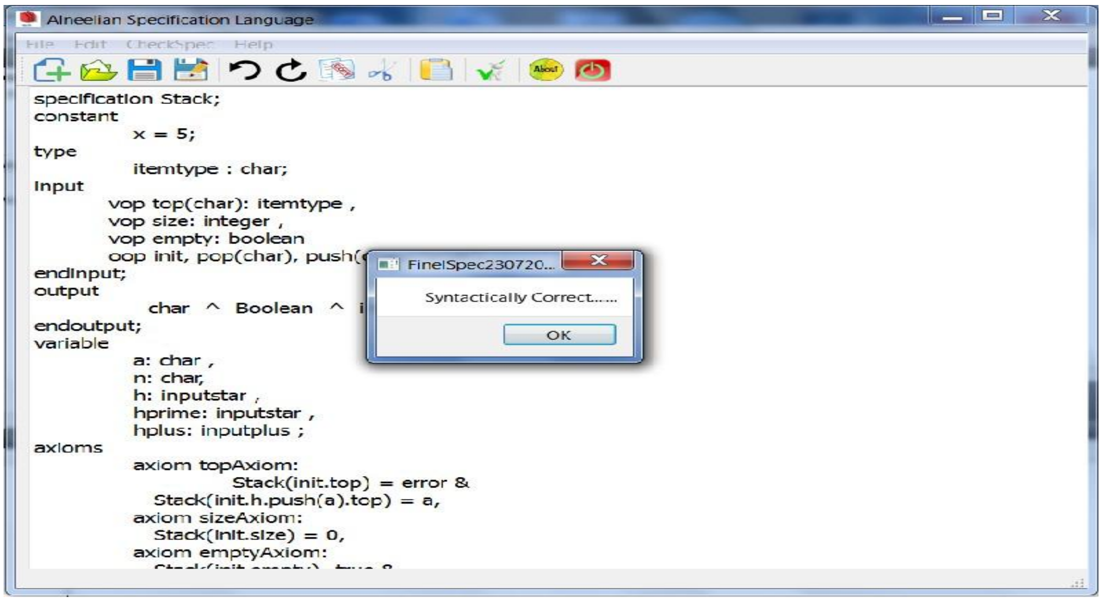
Figure 8.1 shows Alneelain Specification Language ANST in which specification of a stack is appear in the text Editor, it may written directly in text or import from an existing file. As we see in the screen it doesn't show all the specs of the stack, so the remainder of the specification appears in figure 8.2.



```
                    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 8.2: Remainder specification of the stack.

Figure 8.2 shows the remainder specifications of the stack that appear after moving scroll bar.

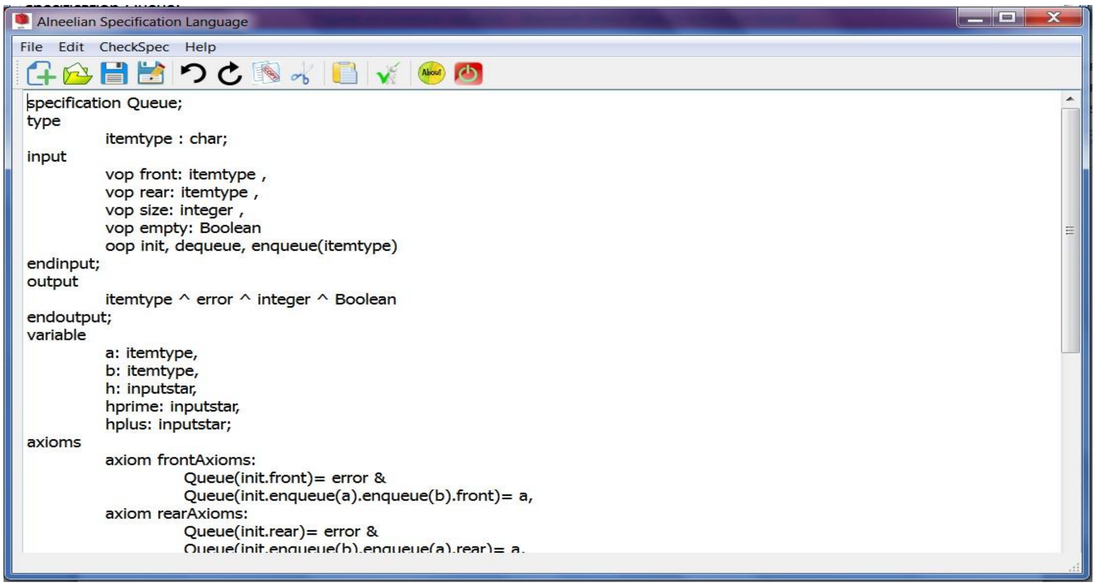


```
specification Stack;
constant
            x = 5;
type
            itemtype : char;
Input
         vop top(char): itemtype ,
         vop size: integer ,
         vop empty: boolean
         oop init, pop(char), push(
endInput;
output
              char  ^  Boolean  ^  i
endoutput;
variable
            a: char ,
            n: char,
            h: inputstar ,
            hprime: inputstar ,
            hplus: inputplus ;
axioms
            axiom topAxiom:
                        Stack(init.top) = error &
              Stack(init.h.push(a).top) = a,
            axiom sizeAxiom:
              Stack(Init.size) = 0,
            axiom emptyAxiom:
```

Syntactically Correct......

Figure 8.3: Checking the stack.

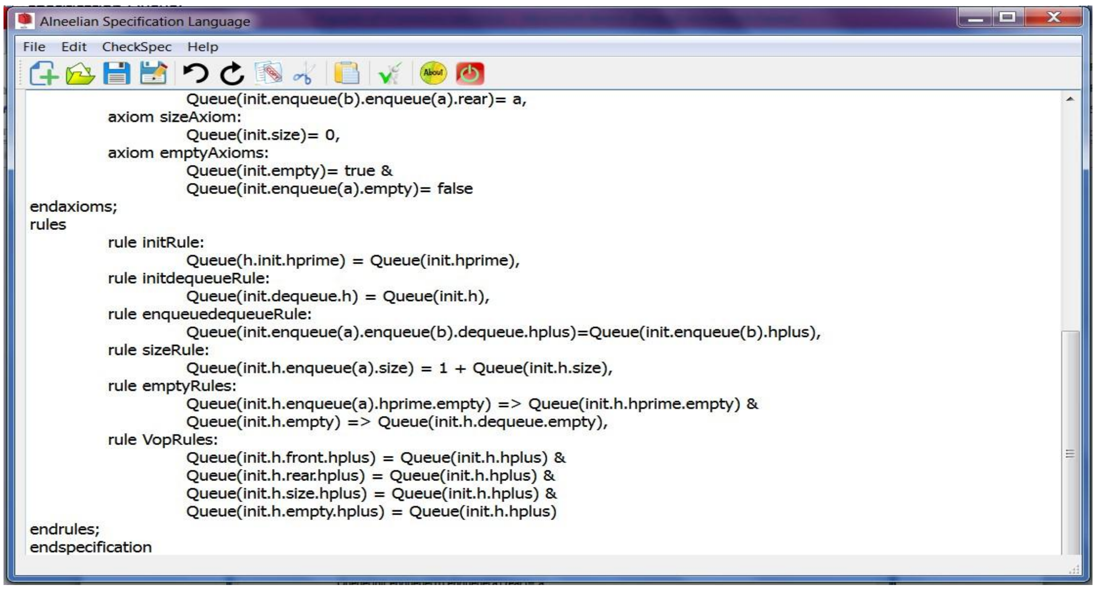
Figure 8.3 is checking whether the stack specifications are syntactically correct.



```
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,
```

Figure 8.4: Queue specification.

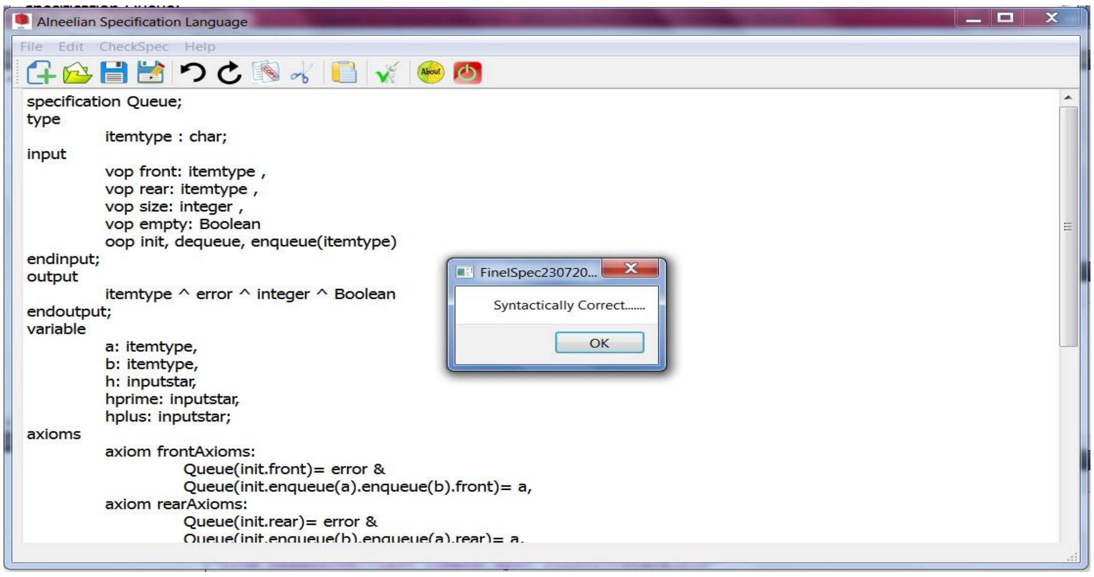
Figure 8.4 shows queue specification. As we see in the screen it doesn't show all the specifications of the queue, so the remainder of specifications appear in figure 8-5.



```
                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
```

Figure 8.5: Remainder specifications of the queue.

Figure 8.5 shows the remainder specifications of the queue that appear after moving scroll bar.



```
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 &
```

Syntactically Correct.......

Figure 8.6: Checking the queue specifications.

Figure 8.6 is checking whether the queue specifications are syntactically correct.

8.3 AN ILLUSTRATIONS OF ALNEELAINVALIDATION TOOL

Here are some windows represent Alneelain Specification Tool NVT

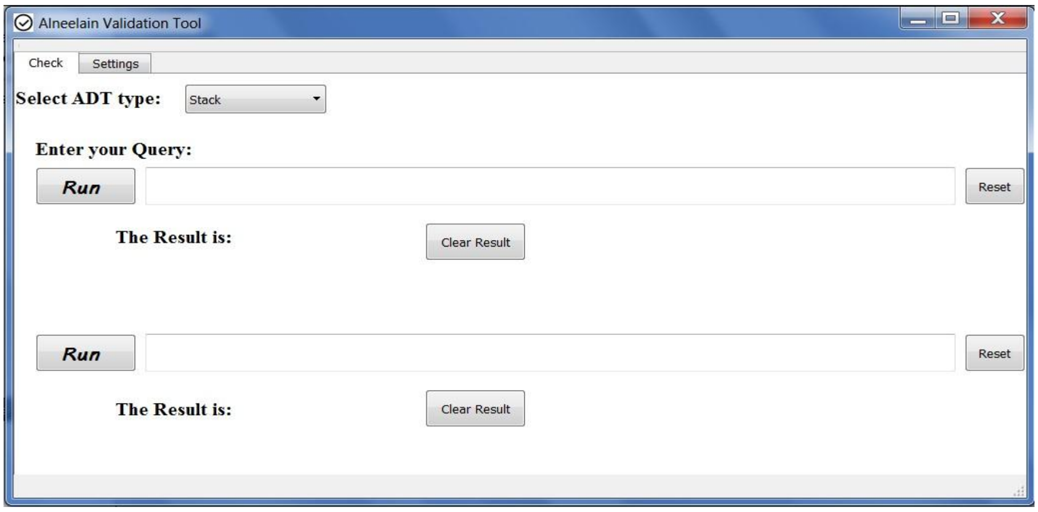
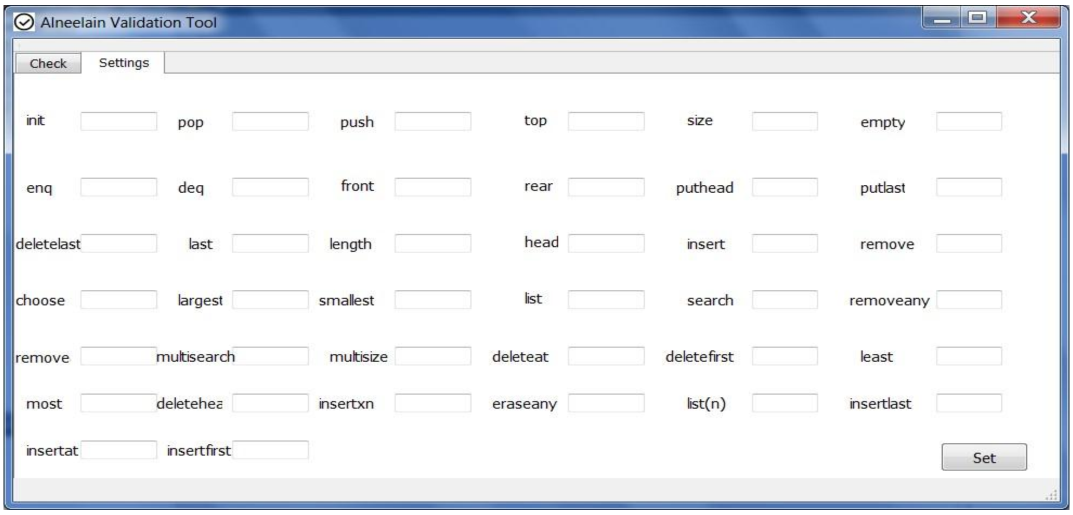


Figure 8.7: AlneelainValidation Tool: Main window.

Figure 8.7 shows the main window of AlneelainValidation Tool.

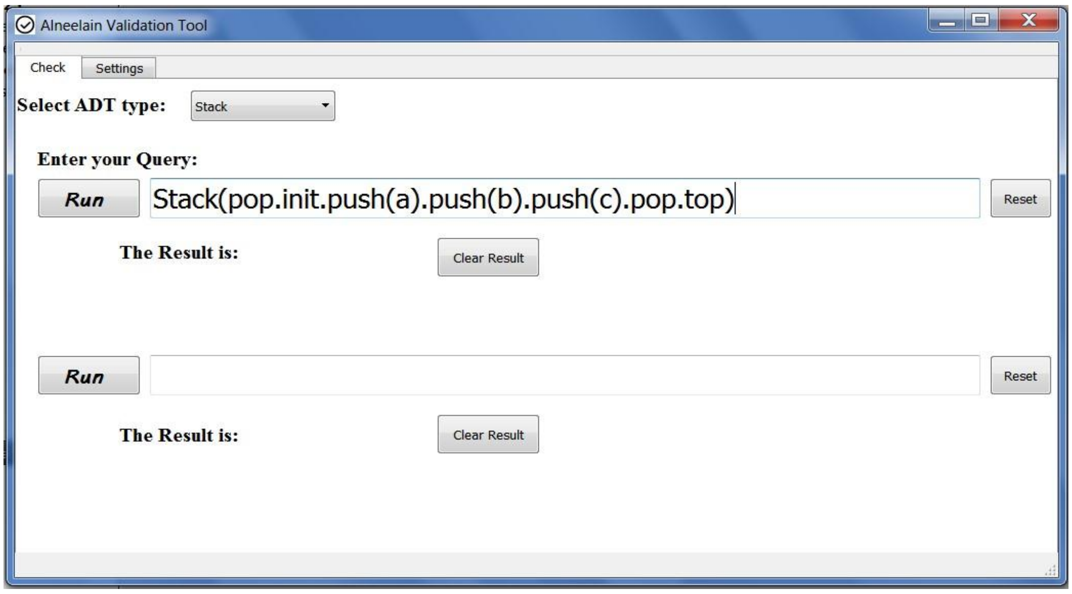

Alneelain Validation Tool

Check | Settings

init		pop		push		top		size		empty	
enq		deq		front		rear		puthead		putlast	
deletelast		last		length		head		insert		remove	
choose		largest		smallest		list		search		removeany	
remove		multisearch		multisize		deleteat		deletefirst		least	
most		deletehea		insertxn		eraseany		list(n)		insertlast	
insertat		insertfirst									

Figure 8.8: Setting tab.

Figure 8.8 shows the setting tab in which a user can change and set the O operations of Abstract Data Types.


Alneelain Validation Tool

Check | Settings

Select ADT type: Stack

Enter your Query:

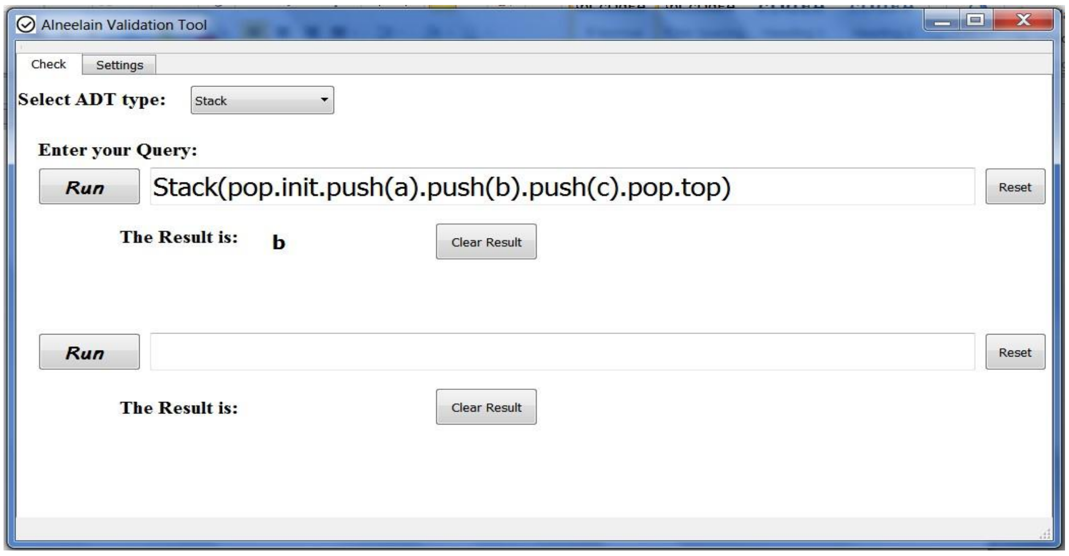
Stack(pop.init.push(a).push(b).push(c).pop.top)

The Result is:

The Result is:

Figure 8.9: checking the top of the stack.

Figure 8.9 shows that the query of Stack(pop.init.push(a).push(b). push(c).pop.top) written in the text box.



Select ADT type: Stack

Enter your Query:

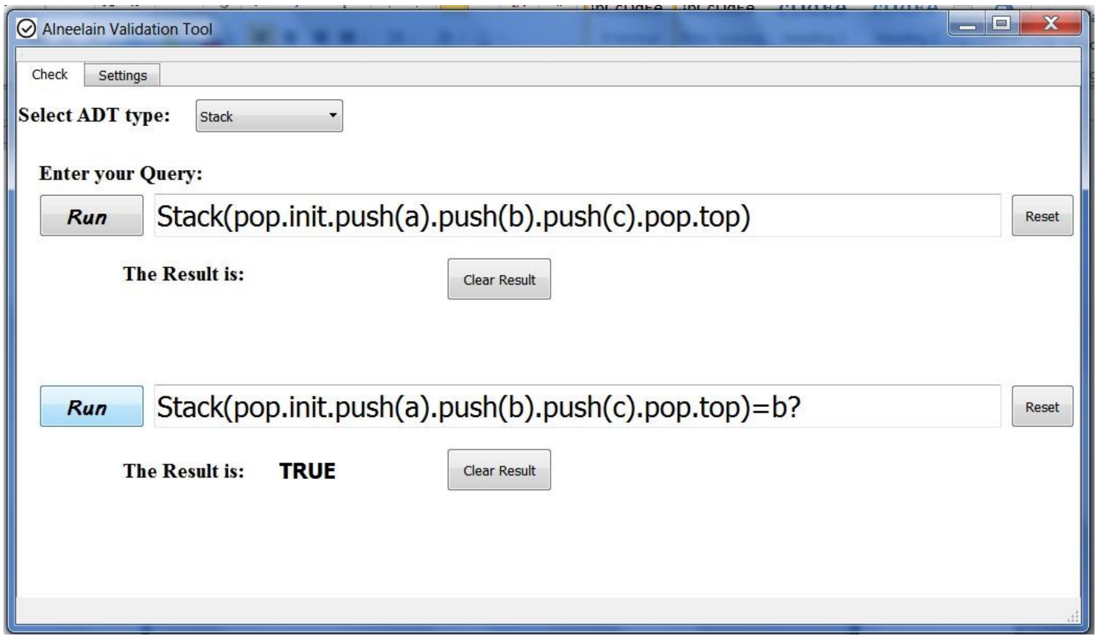
```
Stack(pop.init.push(a).push(b).push(c).pop.top)
```

The Result is: b

The Result is:

Figure 8.10: the result of the top of the stack.

Figure 8.10 shows the result query of Stack(pop.init.push(a).push(b). push(c).pop.top) is b. this result is as same as manual validation that done before using validation tool.



Select ADT type: Stack

Enter your Query:

```
Stack(pop.init.push(a).push(b).push(c).pop.top)
```

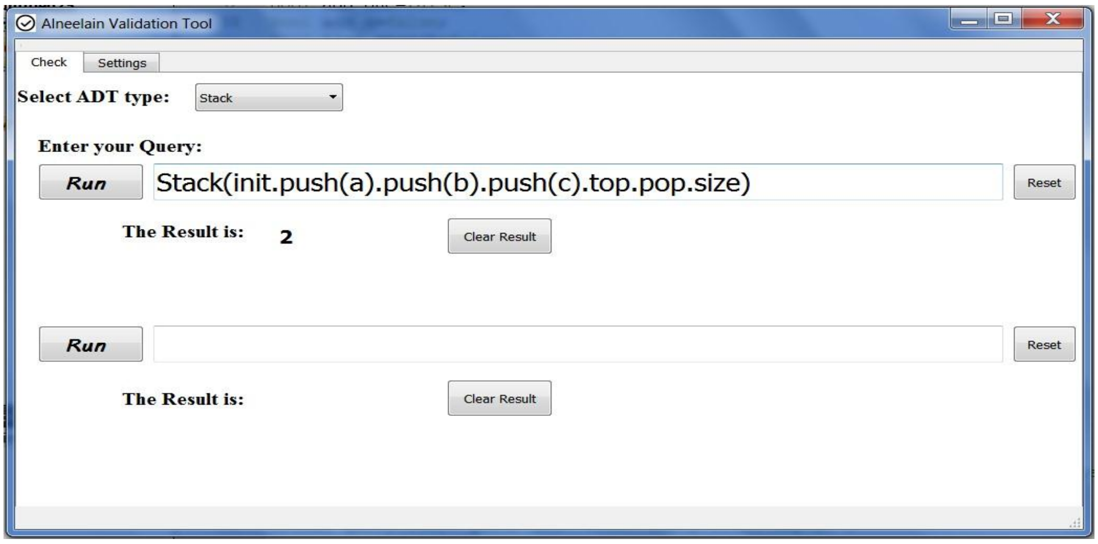
The Result is:

```
Stack(pop.init.push(a).push(b).push(c).pop.top)=b?
```

The Result is: TRUE

Figure 8.11: checking whether the top of the stack is equal to b.

Figure 8.11 shows the previous query of the stack in difference way, which act as theorem prover which it simplify it until get b.



Select ADT type: Stack

Enter your Query:

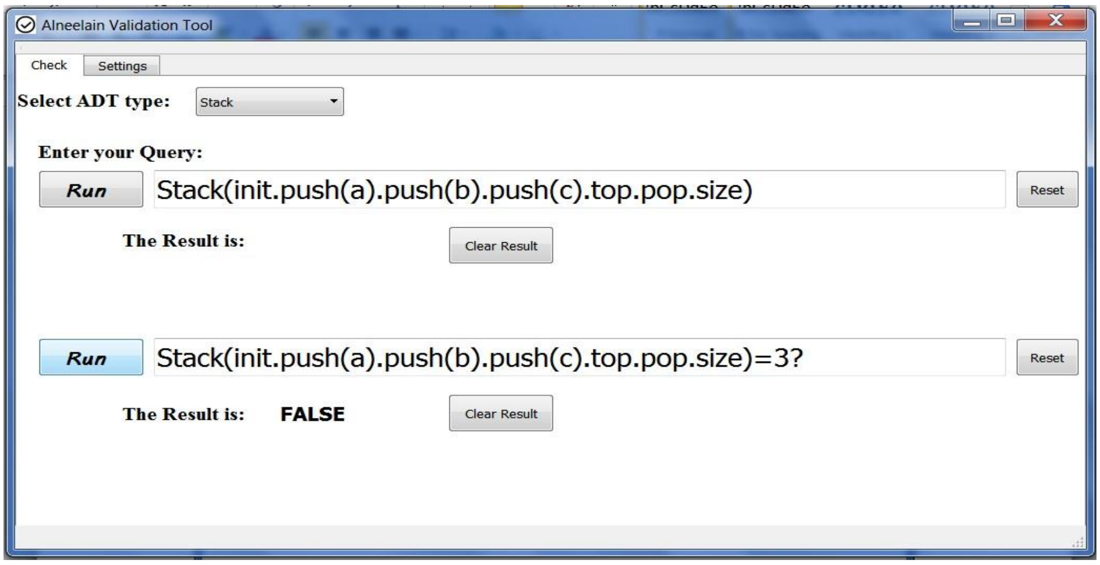
Stack(init.push(a).push(b).push(c).top.pop.size)

The Result is: 2

The Result is:

Figure 8.12: checking the size of the stack.

Check the size of stack that mean how many item in the stack at this moment, figure 8.12 shows the result of query size.



Select ADT type: Stack

Enter your Query:

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

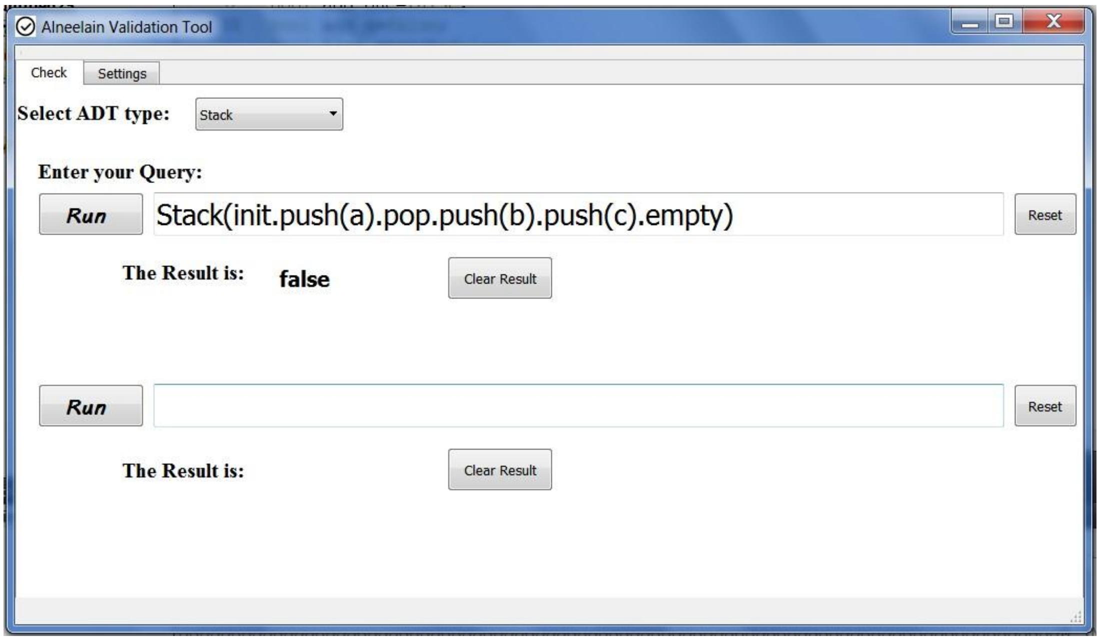
The Result is:

Stack(init.push(a).push(b).push(c).top.pop.size)=3?

The Result is: FALSE

Figure 8.13: checking if the size of the stack is equal to 3.

Figure 8.13 shows that the query it result was 2 in the previous figure 8-24 but the in query appear 3, so the result is false.


Alneelain Validation Tool

Check | Settings

Select ADT type: Stack

Enter your Query:

Run | Stack(init.push(a).pop.push(b).push(c).empty) | Reset

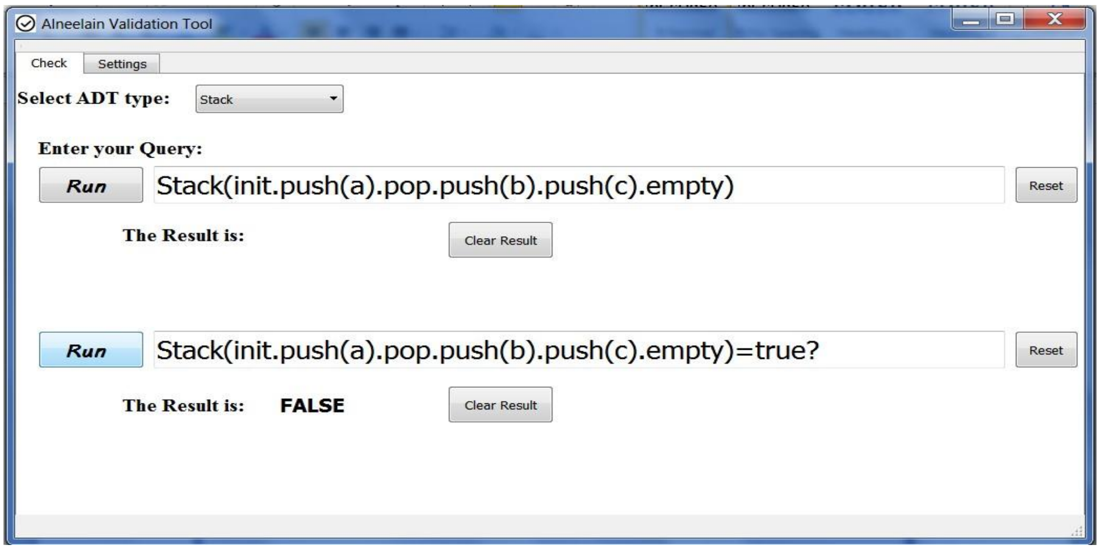
The Result is: false | Clear Result

Run | | Reset

The Result is: | Clear Result

Figure 8.14: checking if the stack is empty.

In figure 8.14 check whether the query: stack(init.push(a).pop.push(b).push(c).empty), the result is false.


Alneelain Validation Tool

Check | Settings

Select ADT type: Stack

Enter your Query:

Run | Stack(init.push(a).pop.push(b).push(c).empty) | Reset

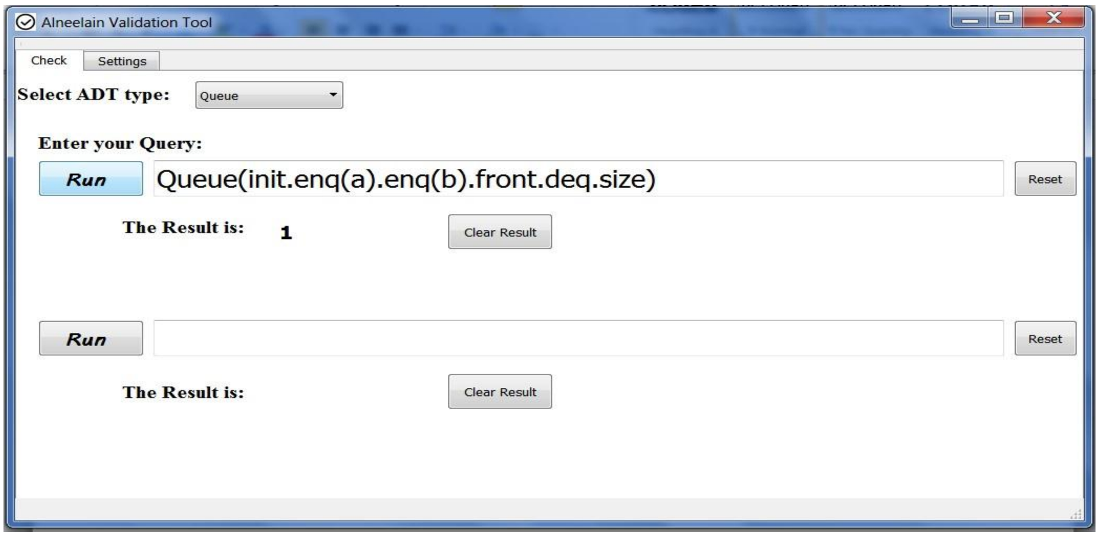
The Result is: | Clear Result

Run | Stack(init.push(a).pop.push(b).push(c).empty)=true? | Reset

The Result is: FALSE | Clear Result

Figure 8.15: checking if the stack is empty.

In figure 8.15 it check whether stack(init.push(a).pop.push(b).push(c).empty)=true, the result is false.



Select ADT type: Queue

Enter your Query:

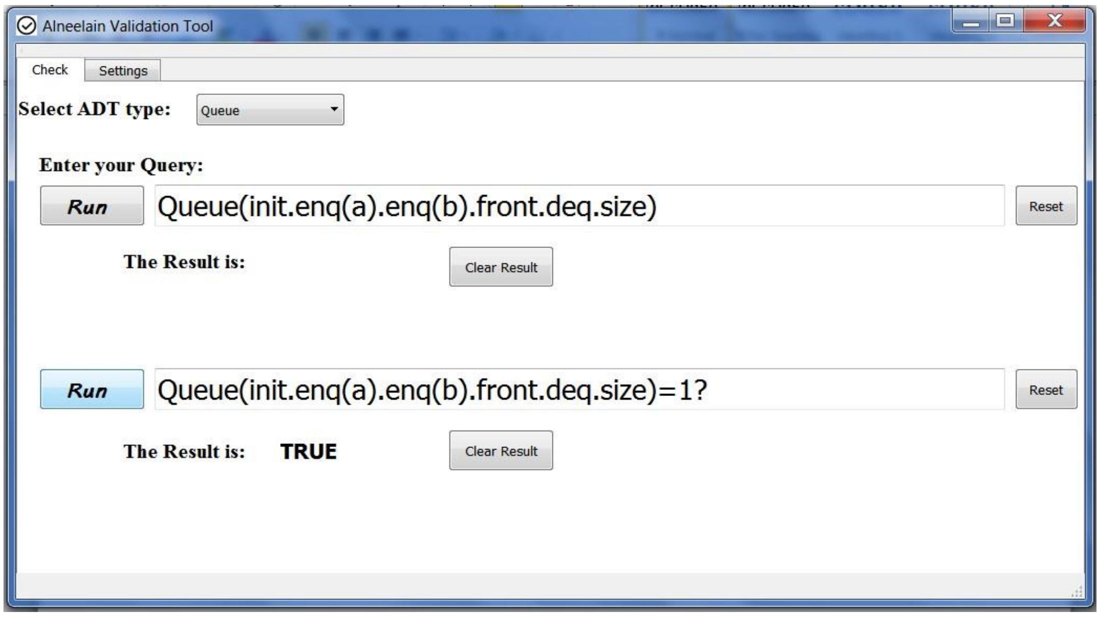
Queue(init.enq(a).enq(b).front.deq.size)

The Result is: 1

The Result is:

Figure 8.16: checking the size of the queue.

Figure 8.16 shows checking the size of the queue, the result is 1.



Select ADT type: Queue

Enter your Query:

Queue(init.enq(a).enq(b).front.deq.size)

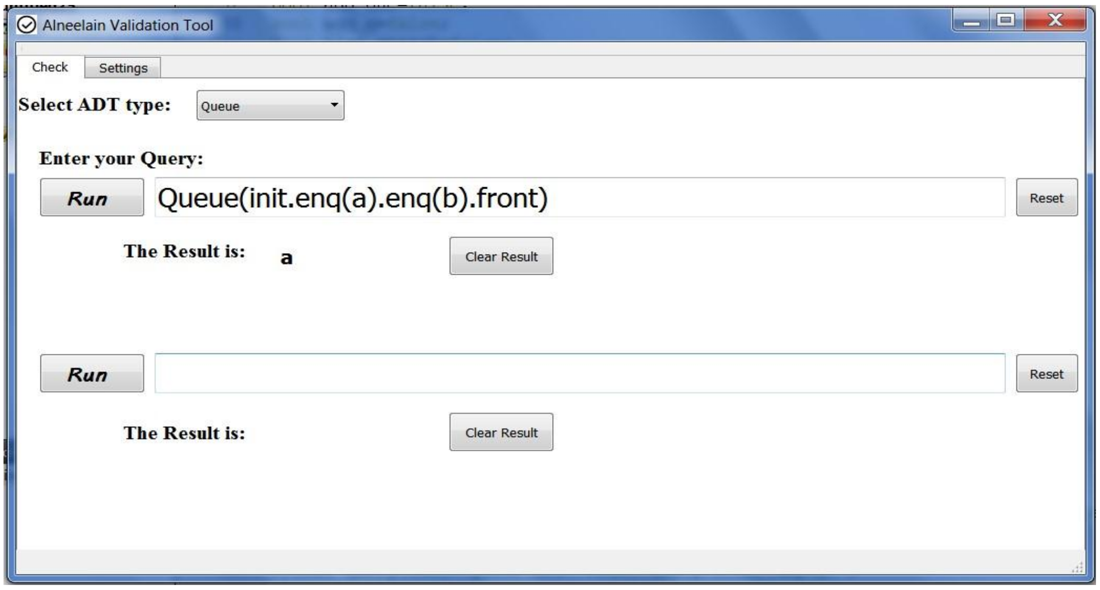
The Result is:

Queue(init.enq(a).enq(b).front.deq.size)=1?

The Result is: TRUE

Figure 8.17: checking if the size of the queue is equal to 1.

Figure 8.17 shows checking whether the size of the queue is 1, the result is true.



Alneelain Validation Tool

Check | Settings

Select ADT type: Queue

Enter your Query:

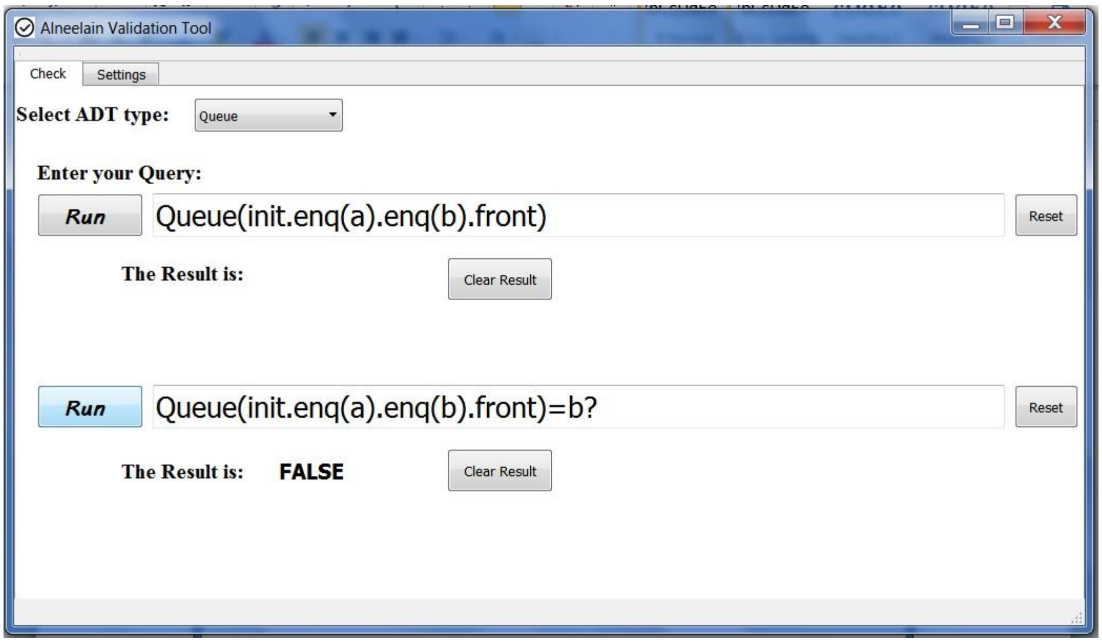
Queue(init.enq(a).enq(b).front)

The Result is: a

The Result is:

Figure 8.18: checking the front of the queue.

Figure 8.18 shows checking the front of the queue.



Alneelain Validation Tool

Check | Settings

Select ADT type: Queue

Enter your Query:

Queue(init.enq(a).enq(b).front)

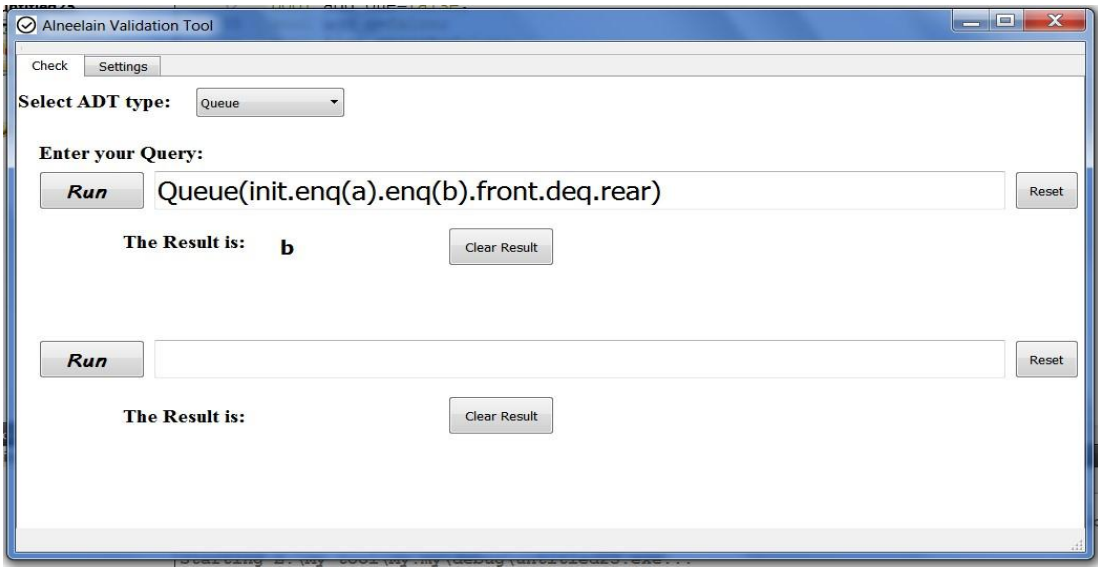
The Result is:

Queue(init.enq(a).enq(b).front)=b?

The Result is: FALSE

Figure 8.19: checking if the front of queue is equal to b.

Figure 8.18 shows the checking if the front of the queue is b, the result is False.



Alneelain Validation Tool

Check | Settings

Select ADT type: Queue

Enter your Query:

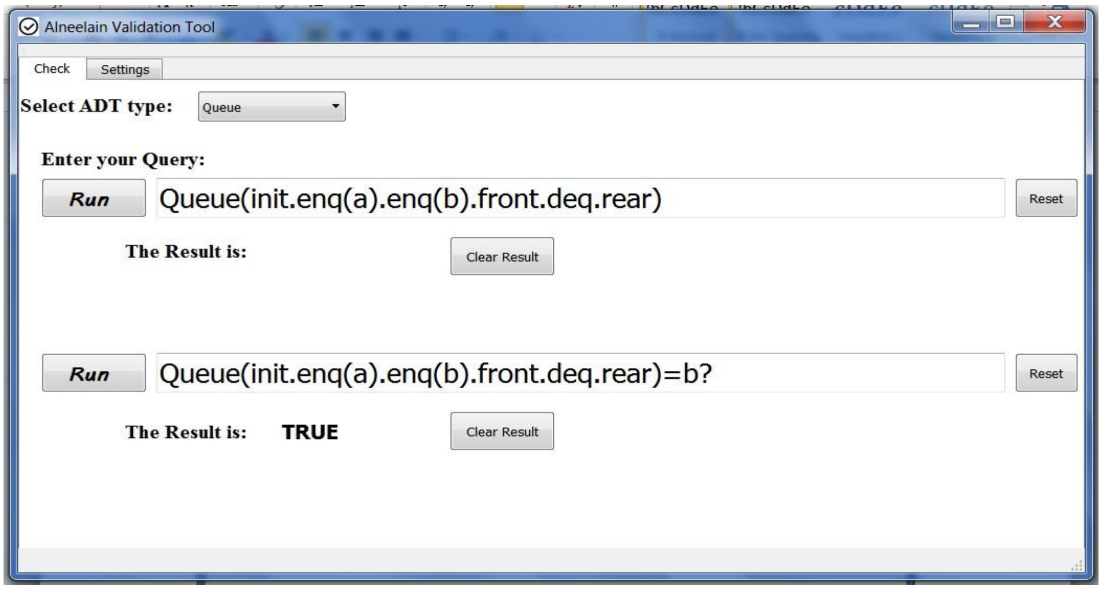
Queue(init.enq(a).enq(b).front.deq.rear)

The Result is: b

The Result is:

Figure 8.20: checking the rear of the queue.

Figure 8.20: checking the rear of the queue, the result is b



Alneelain Validation Tool

Check | Settings

Select ADT type: Queue

Enter your Query:

Queue(init.enq(a).enq(b).front.deq.rear)

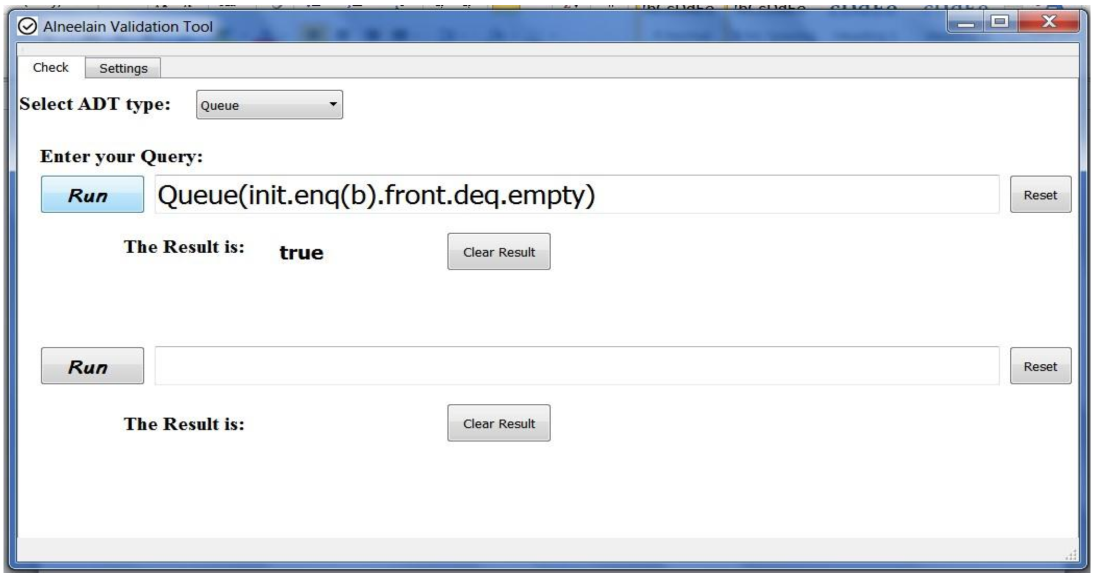
The Result is:

Queue(init.enq(a).enq(b).front.deq.rear)=b?

The Result is: TRUE

Figure 8.21: checking if the rear of queue is equal to b.

Figure 8.21 shows' checking of the rear of the queue is equal to b, the result is true.


Alneelain Validation Tool

Check | Settings

Select ADT type: Queue

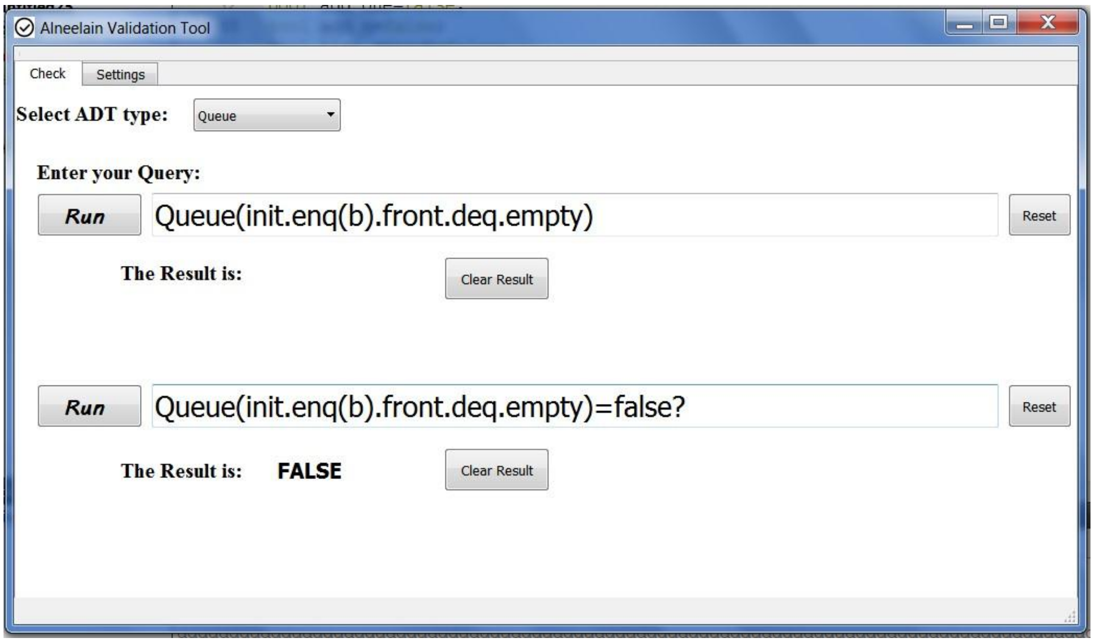
Enter your Query:

```
Queue(init.enq(b).front.deq.empty)
```

The Result is: true

Figure 8.22: checking if the queue is empty.

Figure 8.22 shows checking the empty of the queue, the result is true.


Alneelain Validation Tool

Check | Settings

Select ADT type: Queue

Enter your Query:

```
Queue(init.enq(b).front.deq.empty)
```

The Result is:

```
Queue(init.enq(b).front.deq.empty)=false?
```

The Result is: FALSE

Figure 8.23: checking the empty of the queue equal to false.

Figure 8.23 shows checking the empty of the queue is false, the result is false.

8.4 DEPLOYMENT

In order to execute Alneelain Specification Language and AlneelainValidation Tool the user should access the website to be able to download an open source for both Alneelain Specification Language-NSL and Alneelain validation tool-NVT in the following link: https://sites.google.com/site/abdelrasoulsoftwareengineering/, or, open the website: https://sites.google.com , and then click onto link Software Specification and Validation and then follows these steps:

- Download Alneelain specification language
- Download Alneelain validation tool
- Download and install Qt 5.1.1 2010 32bit for windows
- Download and put a copy of all files that extends with .nln in both in partition D:// and the folder that contain Alneelain specification language
- Download and put a copy of all files that extends with .nln in both in partition D:// and the folder that contain Alneelain validation tool
- Then try to use Alneelain specification language and Alneelain validation tool respectively.

If NSL does not work well a user must download debug and release from website and put them into folder that include NSL.

CHAPTER NINE

CONCLUSION AND FUTURE WORK

9.1 CONCLUSION

In this thesis, we have made our attention in one of the most important software engineering lifecycle phases which is specification phase, and we have introduced a clear challenge which is how to ensure that our specifications of software are valid to correspond to software criteria such as completeness, consistency, and minimality before proceeding to the next phases. Alneelain specification language and Alneelain validation tool were written and developed as main goal of the research.

As a result of the building both of the Alneelain specification language and Alneelain validation tool, we consider that our problem was solved in high percentage, if compared to the results described in Chapter eight, which clearly shows that the validity of the written specifications in the Alneelain specification language.

In order to achieve our goal, we have proposed a behavioral model used to specify abstract data types that can consider as software type and have been proposed a specification language that accommodates and applies this model which this language is used to specify the requirements of abstract data types. A compiler for this language including a lexical analyser and syntax checker has been developed to ensure that any abstract data types that were specified using this language is correct and comply with the rules and structure of this specification language. So Alneelain Specification Language was created.

A validation tool has been developed to be used to validate and ensure that any abstract data type was specified by using Alneelain Specification Language NSL is valid or not valid if a user uses independent validation data so that the AlneelainValidation Tool was created based on rewriting system algorithm. To ensure that the specifications of abstract data types are valid and reflect all the relevant requirements and constraints for the completeness, consistency, and minimality because any faults that arise in this phase it might cause a negative impact on all subsequent phases and increase the cost of a software product, Alneelain Specification Language NSL and Alneelain Specification Tool NVT are developed to be used to in specification and validation abstract data type software before proceeding on to the later phases of developing software.

Many interfaces were presented which illustrate the results after using Alneelain Specification Language to specify abstract data types and after using Alneelain Validation Tool to validate the specification of those abstract data types.

9.2 LIMITATIONS AND FUTURE WORK

There are some limitations to the result of specification and validation shown in this research, Alneelain Specification Language can be used to specify and check any abstract data types that can be written inform of the Alneelain Specification language. however, the validation is done only simple abstract data types such as stack, queue, list, sequence, set, and multiset, etc. whereas, does not validates complicated abstract data types such as tree, table, tree map, and so on. This is due the implementation of rewriting algorithm. There is no a standard method to follow to enable us to rewrite all kinds of abstract data types, because the nature of some complex of those abstract data type. There are many methods for rewriting; first-order logic, and high order logic as well as a random method. However, we cannot implement one method to perform our rewriting or simplification, even we cannot use all at the same time.

As a recommendation, this research opens up unexplored areas to be selected; such as the enhancement of rewriting the algorithm itself, furthermore, building specification language includes validation, verification, and testing to combine into a one universe tool.

REFERENCES

[1] A. P. P. Laplante, what Every Engineer should know about Software Engineering, CRC Press, 2007.

[2] F. Tchierand L. B.A. Rabai, and A. Mili, Ali, “Putting engineering into software engineering: Upholding software engineering principles in the classroom,” *Computers in Human Behavior,* vol. 48, no. 2, pp. 245-254, 2015.

[3] T. Fairouz and R. Latifa Ben Arfa and M. Ali, “Putting engineering into software engineering: Upholding software engineering principles in the classroom,” *Computers in Human Behavior,* vol. 48, pp. 245-254, 2015.

[4] T. Paul P and F. Gary S, “NF-kappa: a key role in inflammatory diseases,” *The Journal of clinical investigation,* vol. 207, no. 1, pp. 7--11, 2001.

[5] G. U. K. S. L. R. a. R. S. F. Marc, “Abstract State Machines, Alloy, B and Z Selected papers from ABZ 2010,” *Science of Computer Programming,* vol. 78, no. 3, pp. 270-271, 2013.

[6] V. M. V. Anneliese, Software engineering: methods and management, Academic Press Professional, Inc, 1990.

[7] M. V. V. Zelkowitz, “Perspectives in software engineering,” *ACM Computing Surveys (CSUR),* vol. 10, no. 2, pp. 197-216, 1978.

[8] B. Barry W, B. John R, and L. Mlity, “uantitative evaluation of software quality,,” in *Proceedings of the 2nd international conference on Software engineering*, Boehm, 1976.

[9] M. Ali and T. Fairouz, Software testing: Concepts and operations, John Wiley & Sons, 2015.

[10] L. Michael , and others, Handbook of software reliability engineering, IEEE computer society press CA, 1996.

[11] H. W. S, Winning with software: An executive strategy, Pearson Education, 2001.

[12] T. Xinming and W. Yingxu and N. Cyprian F, “Formal description of the ADT model of B-Trees,” in *Electrical and Computer Engineering.*, 2005.

[13] B. Boumediene and U. Joseph E, “Direct implementation of abstract data types from abstract specifications,” *IEEE Transactions on Software Engineering*, vol. 5, pp. 649-661, 1986.

[14] G. John V and H. James J, Larch: languages and tools for formal specification, Springer Science & Business Media, 2012.

[15] M. P. M., Modular specification and verification of object-oriented programs, Springer-Verlag, 2002.

[16] B. Dines and H.Martin C, Logics of specification languages, Springer Science & Business Media, 2007.

[17] G. Carlo and M. Dino and M. Angelo, “A logic language for executable specifications of real-time systems,” *Journal of Systems and software*, vol. 12, no. 2, pp. 107--123, 1990.

[18] B. Halimah s, “The Formal Specification for Measuring Dust Concentration Using Z Language,” Universiti Tun Hussein Onn Malaysia, 2014.

[19] I. Sommerville and others, Software engineering, Addison-wesley, 2007.

[20] D. Gannon, John and P. James and Zelkowitz, Marvin V, Software Specification: A Comparison of Formal Methods, Intellect Books, 1994.

[21] B. Bennett, and M. Sitaraman, “Validation of results in testing abstract data types: A method for automation,” in *First Int'l Conf. on Software Quality, Dayton, Ohio*, 1991.

[22] I. J. Hayes, “Using mathematics to specify software,” in *First Australian Software Engineering Conference*, 1986.

[23] C. B. Jones, Systematic software development using VDM, Citeseer, 1990.

[24] J. Spivey, Michael and JR. Abrial, The Z notation, Prentice Hall Hemel Hempstead, 1992.

[25] “Encyclopedia,” A Dictionary of Computing, 20 February 2017. [Online]. Available: http://www.encyclopedia.com. [Accessed 12 may 2018].

[26] H. Klaus-Rudiger, “Open Proof” for Railway Safety Software-A Potential Way-Out of Vendor Lock-in Advancing to Standardization, Transparency, and Software Security,” in *FORMS/FORMAT 2010,* Springer, 2011, pp. 3- -38.

[27] M. Bidoit, R. Hennicker, and M. Wirsing, “Behavioural and abstractor specifications,” *Science of Computer Programming,* vol. 25, no. 2-3, pp. 149-186, 1995.

[28] J. P. Bowen, “Z: A formal specification notation,” in *Software specification methods*, Springer, 2001, pp. 3--19.

[29] G. O'Regan, Mathematical approaches to software quality., Springer Science & Business Media, 2006.

[30] K. Anastasakis,B. Bordbar, G. Georg,and R. ndrakshi, “UML2Alloy: A challenging model transformation,” in *International Conference on Model Driven Engineering Languages and Systems,* 2007.

[31] J.Daniel, “Software Abstractions Logic, Language, and Analysis,” *London, England,* 2006.

[32] J.R. Abrial, The B-book: assigning programs to meanings, Cambridge University Press, 2005.

[33] K Lano, and H. Haughton, “Formal development in B abstract machine notation,” *Information and Software Technology,* vol. 37, no. 5-6, pp. 303-316, 1995.

[34] M. Leonid. and B. Michael, “An approach to combining B and Alloy,” in *International Conference of B and Z Users*, Berlin, Heidelberg, 2002.

[35] B. Manfred, and D. Ernst, Software pioneers: contributions to software engineering, Springer Science & Business Media, 2012.

[36] J. McLean, “A formal method for the abstract specification of software,” *Journal of the ACM (JACM),* vol. 31, no. 3, pp. 600-627, 1984.

[37] M. Frappier,and H. Habrias, Software specification methods: an overview using a case study, Springer Science \& Business Media, 2012.

[38] S. Maalem, and N. Zarour, “Challenge of validation in requirements engineering,” *Journal of Innovation in Digital Ecosystems,* vol. 3, no. 1, pp. 15-21, 2016.

[39] B. Rudolf, and M. Edward, "Organization and maintenance of large ordered indexes," in *Software pioneers*, Springer, 2002, pp. 245-262.

[40] N. Dale, and H. Walker , Abstract data types: specifications, implementations, and applications, Jones & Bartlett Learning, 1996.

[41] S. Kenneth, and K. Barry L, Formal syntax and semantics of programming languages, vol. 340, Addison-Wesley Reading, 1995.

[42] J. Turner, Kenneth and others, Using formal description techniques: an introduction to Estelle, LOTOS and SDL, vol. 85, Wiley New York, 1993.

[43] M. Daniel D. and R. Edwin D, Backus-naur form (bnf, John Wiley and Sons Ltd, 2003.

[44] D. E. Knuth, "Backus normal form vs. backus naur form," *Communications of the ACM,* vol. 7, no. 12, pp. 735--736, 1964.

[45] Y. Levy Jacob, and L. Swee Boon, and others, Compiler with generic front end and dynamically loadable back ends, US Patent 5,812,851: Google Patents, 1998.

[46] D. S. D., "Making compiler design relevant for students who will (most likely) never design a compiler," in *ACM SIGCSE Bulletin,* vol. 34, ACM, 2002, pp. 341--345.

[47] M. B. M., "Towards an Object-Oriented Curriculum," in *TOOLS (11),* Citeseer, 1993, pp. 585-594.

[48] V. Aho, Alfred, and R. Sethi, Ravi, Compilers, Principles, Techniques and Tools, Addison Wesley: Reading, MA, 1986.

[49] "Lexical Analysis. [Online]," Wikipedia contributors, February 2018. [Online]. Available: https://en.wikipedia.org/w/index.php. [Accessed 15 April 2018].

[50] K. P. K., "The Meta-environment.," The Meta-environment., October 2007. [Online]. Available: https://www.metaenvironment.org.

[51] Z. H. Huan, "McMaster University.," 26 september 2007. [Online]. Available: www.cas.mcmaster.ca.

[52] A. M. Sloane, "Software Abstractions: Logic, Language, and Analysis by Jackson Daniel, The MIT Press, 2006, 366pp, ISBN 978-0262101141," *Journal of Functional Programming,* vol. 19, no. 2, pp. 253-254, 2009.

[53] PC. Mag Staff, “Encyclopedia,” Definition of Compiler, 28 February 2007. [Online]. Available: https://en.wikipedia.org. [Accessed 13 3 2018].

[54] H. Daniel, and S. Richard, “Trace specifications: Methodology and models,” *IEEE Transactions on Software Engineering,* no. 9, pp. 1243--1252, 1988.

[55] P. D. L, “On the criteria to be used in decomposing systems into modules,” *Communications of the ACM,* vol. 15, no. 12, pp. 1053--1058, 1972.

appendix A Requirement Description for ADTs

Requirement description of the sequence

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.*

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.*

appendix B Specification of ADTs using ANSL

1-Specification of the queue

```
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-Specification of a sequence

```
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
```

appendix C Manual Validation Data for queue and sequence

1-ADT queue :

queue(init.enq(a).enq(b).front.deq.size) = 1

={ by virtue of V-op rule }

queue(init.enq(a).enq(b).deq.size)

={ by virtue of the second enq deq rule }

queue(init.enq(b).size)

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

1 + queue(init.size)

={ by virtue of the size axiom}

1 +0

= {arithmetic}

queue(init.enq(a).deq.enq(b).enq(c).empty.rear) = c

={ by virtue of the first enq deq rule }

queue(init.enq(b).enq(c).empty.rear)

={ by virtue of V-op rule }

queue(init.enq(b).enq(c).rear)

={ by virtue of the second rear axiom, with h = < enq(b) >}

c

queue(init.enq(a).enq(b).front.init.deq.enq(a).deq.size) = 0

={ by virtue of init rule }

queue(init.deq.enq(a).deq.size)

={ by virtue of init deq rule }

queue(init.enq(a).deq.size)

={ by virtue of the first enq deq rule }

queue(init.size)

={ by virtue of size axiom }

0

queue(init.enq(a).enq(b).front.init.deq.enq(a).deq.empty.empty)= true

={ by virtue of init rule }

queue(init.deq.enq(a).deq.empty.empty)

={ by virtue of init deq rule }

queue(init.enq(a).deq.empty.empty)

={ by virtue of V-op rule }

queue(init.enq(a).deq.empty)

={ by virtue of the first enq deq rule }

queue(init.empty)

={ by virtue of the first empty axiom }

true

queue(init.enq(a).enq(b).front.init.deq.deq.size.enq(c).size.front) = c

={ by virtue of init rule }

queue(init.deq.deq.size.enq(c).size.front)

={ by virtue of V-op rule }

queue(init.deq.deq.enq(c).front)

={ by virtue of init deq rule, applied twice }

queue(init.enq(c).front)

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

c

queue(init.enq(a).enq(b).front.init.deq.deq.init.size.front) = error

={ by virtue of init rule, applied twice}

queue(init.size.front)

={ by virtue of V-op rules }

queue(init.front)

={ by virtue of the first front axiom }

error

2-ADT Sequence:

sequence(init.puthead(a).puthead(b).putlast(c).deletelast.puthead (d).empty) = False

={ by virtue of the putlast deletelast rule }

sequence(init.puthead(a).puthead(b).puthead (d).empty)

={ by virtue of the first empty rule , with h=<puthead(a).puthead(b) > ,h'=<> }

sequence(init.puthead(a).puthead(b).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

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 first length rule , with h=<> }

1+ sequence(init.length)

={ by virtue of the length axiom}

+ 0 = {arithmetic}

1

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 , with h=< putlast(c) > }

d

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

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 puthead deletehead rule, with h = <putlast(c) > }

sequence(init.putlast(c).last)

={ by virtue of the fifth last axiom }

c

sequence(init.puthead(a).puthead(b).puthead(c).deletehead.deletehead.deletehead.puthe ad(d).putlast(e).deletelast.head) = d

={ by virtue of the puthead deletehead rule ,applied three times }

sequence(init.puthead(d).putlast(e).deletelast.head)

={ by virtue of the putlast deletelast rule, with h = < puthead(d) >}

sequence(init.puthead(d).head)

={ by virtue of the fourth head axiom}

d

sequence(init.puthead(a).head.puthead(b).length.puthead(c).last.deletehead.deletehead.empty.deletehead.length) = 0

={ by virtue of V-op rules }

sequence(init.puthead(a).puthead(b).puthead(c).deletehead.deletehead.deletehead.length)

={ by virtue of the puthead deletehead rule ,applied three times }

sequence(init.length)

={ by virtue of the length axiom}

0

sequence(init.puthead(a).puthead(b).puthead(c).deletehead.deletehead.deletehead.deletelast.empty)   = true

={ by virtue of the puthead deletehead rule ,applied three times }

sequence(init.deletelast.empty)

={ by virtue of the init deletelast rule}

sequence(init.empty)

={ by virtue of the first empty axiom}

true