University of Durham
Department of Computer Science

The Construction of Oracles for Software Testing

M.Sc. Thesis

Xiaodong Zhang

1993

Abstract

Software testing is important throughout the software life cycle. Testing is the part of the software development process where a computer program is subject to specific conditions to show that the problem meets its intended design. Building a testing oracle is one part of software testing. An oracle is an external mechanism which can be used to check test output for correctness. The characteristics of available oracles have a dominating influence on the cost and quality of software testing. In this thesis, methods of constructing oracles are investigated and classified. There are three kinds of method of constructing oracles: the pseudo-oracle approach, oracles using attributed grammars and oracles based on formal specification.

This thesis develops a method for constructing an oracle, based on the Z specification language. A specification language can describe the correct syntax and semantics of software. The contextual part of a specification describes all the legal input to the program and the semantics part describes the meaning of the given input data. Based on this idea, an oracle is constructed and a prototype is implemented according to the method proposed in the thesis.

Acknowledgements

I wish to thank Dr. D. J. Robson for kindly agreeing to be my supervisor, and for his supervision and administration throughout the research project.

I would also like to thank my husband Hongji Yang for his encouragement, unselfish support and beneficial technical advice.

I am feeling in debt to my son Tianxiu Yang for being unable to look after him wholeheartedly whilst undertaking this study.

Copyright

Contents

List of Figures

Chapter 1

Introduction

1.1 Purpose of the Research

Program proofs and program testing are two methods used to evaluate program "correctness". Program proof assures the correctness of a program using a mathematical method of verifying the logic and the function of a program or program part. Proofs are expensive and minor changes in a program may require extensive change in its proof.

Program testing is the process of evaluating a program, with or without execution, to verify that it satisfies specified requirements. Although it is thought that testing can only expose errors of program and not demonstrate their absence [32], program correctness is most often evaluated via this technique. In the field of software engineering, testing generally contains four tasks:

1. Select a set of input data according to a suitable testing strategy

2. Obtain actual output data by executing program on input data

3. Derive expected output data

4. Compare the actual output results and expected output results, in order to validate the program.

In the testing literature, the third and fourth tasks are known as the oracle problem. Oracles are an external source which, for any given input description, can provide a complete description of the corresponding output behaviour (ie. expected outputs) and determine whether the test output conforms with the expected output. Generally, a programmer serves as a test oracle, calculating expected output data from the specification by hand, comparing actual outputs with their expected outputs and agreeing too eagerly with the results of program execution.

A definition of the expected output or result is a necessary part of the software testing. This obvious principle is one of the most frequent mistakes in software testing and it is something that is based on human psychology. If the expected result of a test has not been predefined, chances are that a plausible, but erroneous result will be interpreted as a correct result[32]. Therefore, more attention should be paid to deriving the expected output of test data. Properly exploited, an automated oracle can improve not only testing productivity but also its efficacy.

In this thesis, the construction of an automated testing oracle is proposed, and this is an alternative to a human oracle. This automated oracle integrates formal specification techniques with the process of software testing. The greatest advantage of this technique is the generation of an oracle which functions independently of human decisions. This provides a strong foundation upon which to build a complete testing system. The generation of a complete testing system from a formal specification would provide a greatly enhanced tool.

1.2 Scope of the Thesis

The central aim of the research described in this thesis is to look at methods for constructing an automated testing oracle. The work is divided into two parts:

1. the development of a testing oracle construction method, and

2. the design and implementation of a prototype automated oracle based on the Z specification language.

In a formal specification, a context-free grammar can be defined with the addition of synthesised and inherited attributes. The attributes provide the information required to generate meaningful test cases. Input test data points and the corresponding expected outputs are then generated either randomly or systematically from the grammar. Following this idea, a construction method of an automated oracle is proposed and implemented based on the Z formal specification language. The system consists of two basic components: a parser to generate the parse tree of the Z specification in Lex and YACC, and an interpreter to generate expected outputs of the software in the C programming language.

The scope of this work has been limited to simple and small specifications written in the Z specification language. It is not intended to produce a tool for large complex Z specifications which is an issue for future research.

1.3 Thesis Structure

Chapter 2, **Software Engineering and Testing**, gives the definition of software engineering and testing and describes the importance of testing in software engineering.

Chapter 3, **Software Testing**, provides an overview of software testing, introduces and evaluates techniques in software testing, and shows that testing is necessary at all stages of the software life cycle.

Chapter 4, **Testing Oracles**, surveys related research in testing oracle, gives definitions of testing oracles, and introduces testing techniques and current research in the area of testing oracles.

Chapter 5, **Construction of an Automated Oracle**, proposes a method of constructing an automated testing oracle based on the Z specification language, and introduces theories, concepts and tools related to constructing an automated testing oracle.

Chapter 6, **Implementation**, describes the implementation of the prototype components designed in chapter 4.

Chapter 7, **Result and Evaluation**, presents an example of the result of using the system and evaluates the research in the thesis.

Chapter 8, **Conclusions**, summarises the thesis, assessing the research carried out in thesis against the proposed criteria and suggesting areas for future research.

Chapter 2

Software Engineering and Testing

2.1 Definition of Software Engineering

As software systems have grown more sophisticated and complex, software developers have sought new methods for their development. Software engineering is a response to that need.

The IEEE standard Glossary of Software Engineering terminology (IEE83) defines *software engineering* as: "The systematic approach to the development, operation, maintenance, and retirement of software", where software is defined as: "Computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system."

From this definition, we know that software engineering is the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates [16].

Software engineering is a new technological discipline distinct from, but based on the foundations of, computer science, management science, economics, communication skills, and the engineering approach to problem solving.

Because software engineering is concerned with the development and maintenance of technological products, problem-solving techniques common to all engineering disciplines are utilised. Engineering problem-solving techniques provide the basis for project planning, project management, systematic analysis, methodical design, careful fabrication, extensive validation, and ongoing maintenance activities. Appropriate notations, tools and techniques are applied in each of these areas. Furthermore, engineering, scientific principles, economics, and communication skills are combined within the framework of engineering problem solving. The result is software engineering.

The term "*computer software*" is often taken to be synonymous with "computer program" or "source code". Rigorously, "computer software" is synonymous with "software product". Thus computer software includes the source code, and all the associated documents and documentation that constitute a software product. Requirement documents, design specifications, source code, test plans, principles of operation, user manuals, installation instruction, and training aids are all components of a software product. Software products include system-level software as well as application software developed to solve specific problems for end users of computing systems [16]. But in this thesis, 'software' just means a 'program'.

2.1.1 The Software Life Cycle

Like all other large-scale systems, large software systems take a considerable time to develop and are in use for an even longer time. A number of distinct stages in this period of development and usage must be identified. The most commonly used stages in the software development life cycle are problem definition, requirement analysis, specification, design, testing, and operation and maintenance [21] [47]. The waterfall model of software life cycle is illustrated in Figure 2.1. Software development cascades from the highest phase to more detailed implementations.

1. *Problem definition*

   The goal of problem definition is to define the problem in user terms as precisely as possible. Problem definition also helps the user to better understand the problem.

   The result of this phase is a document that defines the problem in terms of the user's objectives and major constraints.

2. *Requirement Analysis*

   The requirement analysis represents a period of interaction between the user and the analyst, the latter being the computer professional assigned to work with the user during this phase. The original requirements are examined and tested for internal consistency. In other words, any contradictions or ambiguities in the requirements are discussed with the user until they are resolved to the satisfaction of both parties. The requirements are then refined until the user and analyst are in complete agreement as to the expected detailed behaviour of the new software.

3. *Specification*

   The objectives of the specification phase are to describe what the solution looks like:

   - what input the system is going to process
   - what function it will perform for each input

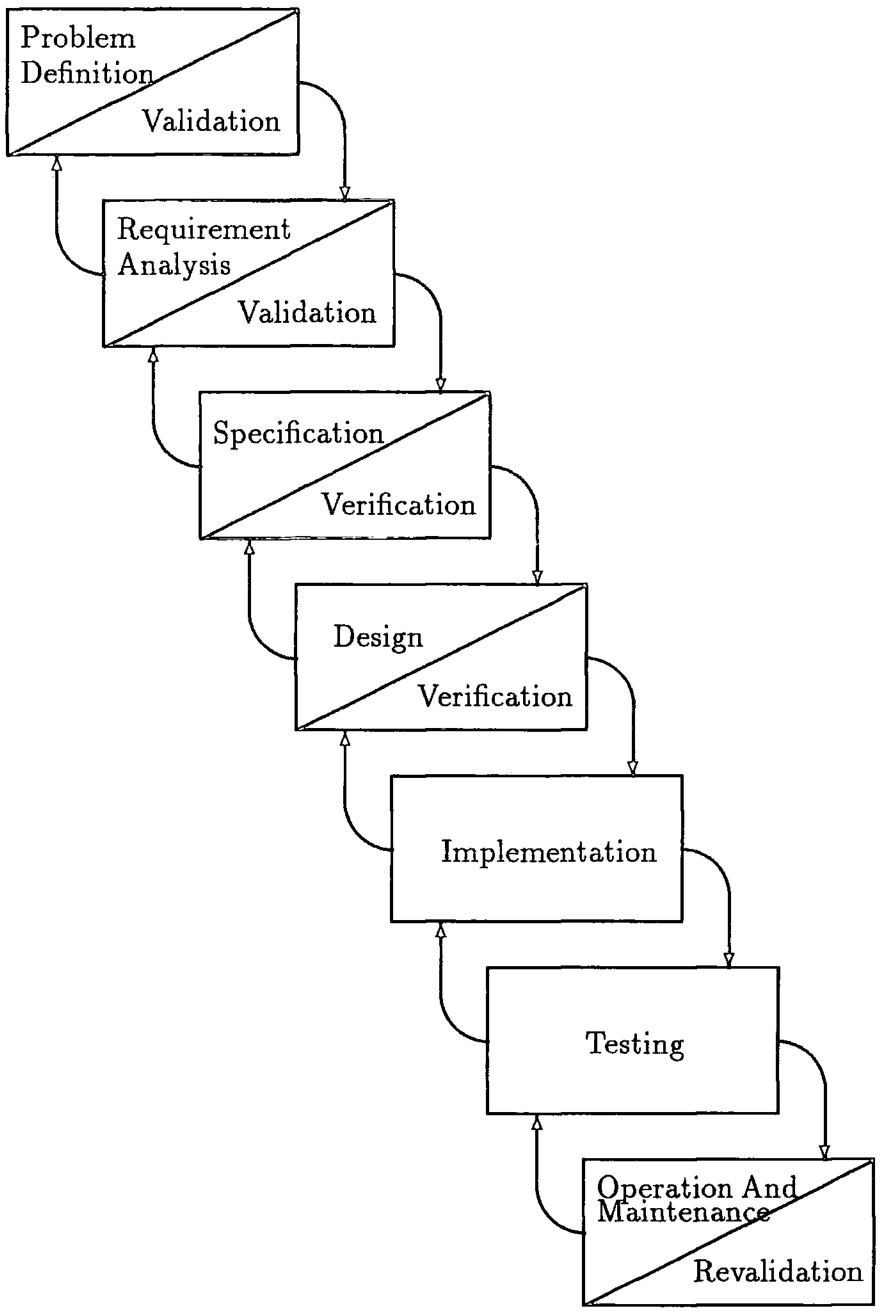

Figure 2.1: The Waterfall Software Life Cycle Model

- what the corresponding output will be, and
- whether the specified system meets the requirements and whether its further development satisfies to the project plan or whether the project plan must be altered.

A specification of a program may serve different purposes [26]:

- Specifications are used for program documentation.
- Specifications serve as a mechanism for generating questions. The construction of specifications forces the designers to think about the requirement definition and the intrinsic properties and functionalities of the software system to be designed.
- A specification can be considered as a kind of contract between the designers of a program and its customers.
- Specifications are a powerful tool in the development of a program during its software life cycle. The presence of a good specification helps not only designers, but also implementors and maintainers.
- With regard to program validation, specifications may be very helpful to collect test cases to form a validation suite for the software system.

4. *Design*

With the commencement of the design stage the attention of software developers focus on the question of how the user's requirements are to be implemented. This means that ideas on the structure of the programs and the data structures on which they will work are generated, and the best ideas are selected for further development. Design does not involve a completely systematic way of working backward from the requirements. It requires iteration, synthesis and analysis.

In the process of software design, there are three distinct types of activities: external design, architectural design and detailed design. Architectural design and detailed design are collectively referred to as internal design.

External design of software involves conceiving, planning out, and specifying the externally observable characteristics of a software product. These characteristics include user displays and report formats, external data source and data sinks, and the functional characteristics, performance requirements, and high level process structure for the product. External design begins during the analysis phase and continues into the design phase.

Internal design involves conceiving, planning out, and specifying the internal structure and processing details of the software product. The goals of internal design are to specify internal structure and processing details, to record design decisions and indicate why certain alternatives and trade-offs were chosen, to elaborate the test plan, and to provide a blueprint for implementation, testing, and maintenance activities. The work products of internal design include a specification of architectural structure, the details of algorithms and data structures, and the test plan.

5. *Implementation*

The implementation phase of software development is concerned with translating design specification into source code. The primary goal of implementation is that debugging, testing, and modification are eased. This goal can be achieved by making the source code as clear and straightforward as possible. Simplicity, clarity, and elegance are hallmarks of good programs;

obscurity, cleverness and complexity are indications of inadequate design and misdirected thinking.

6. *Testing*

   The written code should be tested rigorously based on the required quality characteristics. This is the objective of the test phase. Testing usually proceeds in several steps. As soon as code has been written, it should be tested. First pieces (modules) are tested in isolation. This is called *unit testing.*

   Later, modules are tested in groups to see whether they interact properly. This is called *integration testing.* In most instances, the newly developed software system must be tested in its actual running environment. If there are several environments, each must be tested in what is called *system testing.*

   Software testing is important throughout the software life cycle. It can be seen that some form of testing is necessary at all stage of the life cycle (Figure 2.1). The terms "validation" and "verification" can be looked upon as various forms of testing; they are defined fully in Section 2.3.

7. *Operation and Maintenance*

   Normally this is the longest life cycle phase [40]. The system is installed and put into practical use. The maintenance phase focuses on *change* that is associated with error corrections, adaptations required as the software's environment evolves, and modifications due to enhancements brought about by changing customer requirements. The maintenance phase reapplies the steps of the definition and development phases, but does so in the context of existing software. Three types of change are encountered during the maintenance phase:

Correction. Even with the best quality assurance activities, it is likely that the customer will discover defects in the software.

Adaptation. Over time the original environment (e.g., CPU, operating system, peripherals) for which the software was developed is likely to change.

Enhancement. As software is used, the customer/user will recognise additional functions that would provide benefit.

2.2 Software Engineering and Testing

The software engineering life cycle shows that some form of testing is carried out throughout the life time of a software product. It can be seen that some form of testing is necessary at all stages. Testing typically consumes an enormous proportion (sometimes as much as 50 %) of the effort of developing a system [4] therefore we need to improve techniques to tackle the problem.

Software testing is part of the validation process which is normally carried out during implementation and also in a different form, when implementation is complete. Testing involves exercising the program using data, observing the program outputs and inferring the existence of program errors or inadequacies from anomalies in that output.

Testing requires a test plan that describes what is to be tested and when and how it is to be tested. In its most detailed form, the test plan includes a specification of the test cases and the expected outputs.

Although only a part of the overall validation process, program testing is the only technique used to validate a program in most programming organisations. Program verification is a widely used validation technique. After testing, the software system is delivered to the customer. Some testing techniques are described in detail in the next section.

2.3 Definition of Software Testing

Testing is a term with varied meanings, there are other related terms, such as validation, verification, certification, and debugging. It is important to give a clear definition of terms used in software testing.

In this thesis, the following definitions will be used for testing, validation, verification, and debugging [31,40]:

- *Testing* — Testing is the process of evaluating a program, with or without execution, to verify that it satisfies specified requirements. It is the process of feeding sample input data into a program, executing it, and inspecting the output and/or behaviour for correctness. Testing is exercising different modes of a computer program's operation through different combinations of input data (test cases) to find errors.

- *Validation* — Validation is the process of checking that system and its structure as implemented meets the specifications of the user ('Are we implementing the right product?'). The process includes ensuring that specific program functions meet their requirements and specification. Validation also includes the prevention, detection, diagnosis, recovery and correction of errors. Validation is more difficult than the verification process since it involves questions about the completeness of the specification and environment information. The validation of a software is a continuing process through each stage of the software production.

- *Verification* — Verification is the process of ensuring that the system and its structure meet the functional requirements of the baseline specification document ('Are we implementing the product right?'). Verification is usually only concerned with the software's logical correctness (i.e., satisfying the functional requirements) and may be a manual or a computer based process (i.e., testing software by executing it on a computer). Now, the post-implementation validation and verification technique rely on program testing.

- *Certification* — Certification extends the processes of verification and validation to an operational environment; confirms that the system is operationally effective; is capable of satisfying requirements under specified operating conditions; and finally guarantees its compliance with requirements in writing. Certification usually implies the existence of an independent quality control group for the acceptance testing of the overall system. The acceptance testing may be accomplished by operational testing, and/or placing the system in simulated operation. Certification is the formal demonstration of system acceptability to obtain authorisation for its operational use.

- *Debugging* — Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. The debugging process will always have one of two outcomes: (1) the error (cause of the system) will be found, corrected, and removed, or (2) the error will not be found.

An *error* is a mistake by a programmer or designer. It may result in textual problem with the code called a fault. A *failure* occurs when a program computes an incorrect output for an input in the domain of the specification [33].

A *test case* is a detailed design, consisting of both the required input data for program execution, and a precise description of the correct output of the program for that set of input data.

The term *software testing* is often used to describe techniques of checking software by executing it with input data. A wider meaning would be: testing includes any techniques of checking software, such as symbolic execution and program proving as well as the execution of test cases with data. The expected outputs derived by tester are based on the specification derived manually.

Chapter 3

Review of Software Testing

3.1 Strategies of Software Testing

There are many widely differing testing techniques, but for all the apparent diversity they cluster or separate according to test case design. A test case is a detailed design, consisting of both the required input data for program execution, and a precise description of the correct output of the program for that set of input data.

Selecting test input data, like other processes that attempt to make the best choices, must be based on all available information of facts rather than coincidence, myth or guesswork. There are a number of sources of information about a program or a program unit. The source code is a source of information, if it is available to the testers. Another main source of information is the functional specification of the program or module. These two main sources of information give rise to two main streams of testing approaches, *structural testing* and *functional testing.*

Once a strategy or a combination of strategies is decided, there are a number of established techniques which can be followed to design test cases. Some of these techniques require the execution of the program and some do not. Therefore, testing techniques can also be classified as either dynamic or static. *Static analysis* is any testing technique that does not involve the execution of the program under test. *Dynamic analysis* is any testing techniques that requires the program to be executed.

3.2 Functional Testing

A testing strategy may be based upon one of two starting points: either the specification or the software is used as the basis for testing. Starting from the specification the required functions are identified. The software is then tested to assess whether they are provided. This is known as *functional testing.*

Functional testing involves two main steps. Firstly, it is to identify the functions which the software is expected to perform. Secondly, it is to create test data which will check whether these functions are performed by the software. No consideration is given to how the program performs these functions.

Functional testing has been termed a *black box* approach as it treats the program as a box with its contents hidden from view. The tester submits test cases to the program based on their understanding of the intented function of the program. An important component of functional testing is an *oracle.* An *oracle* is someone which can state precisely what the output of a program execution will be for a particular test data. Useful techniques in performing functional testing include[12]:

- Random testing
- Adaptive perturbation testing
- Cause-effect graphing testing

Random Testing Random testing produces test data without reference to the code or the specification. The main software tool required is a random number generator. Potentially, there are some problems for random testing. Most significantly it may seem that there is no guarantee for a complete coverage of the program. For example, when a constraint on a path is an equality e.g. A= B+5 the likehood of satisfying this constraint by random generation is low. Alternatively, if complete coverage is achieved then it is likely to have generated a large number of test cases. The checking of the output from the execution would require an impractical level of human effort.

Adaptive Perturbation Testing This technique is based on assessing the effectiveness of a set of test cases. The effectiveness measure is used to generate future test cases with the aim of increasing the effectiveness.

The cornerstone of the technique is the use of executable assertions which the software developer inserts into the software. An assertion is a statement about the reasonableness of values of variables. The aim is to maximise the number of assertion violations recorded. Each test case is now considered in turn. The single input parameter of the test case that contributes least to the assertion violation count is identified. Optimisation routines are then used to find the best value to replace the discarded value such that the number of assertions is maximised.

Cause-effect Graph Testing The strength of cause-effect graphing lies in its power to explore input combinations. The graph is a combinatorial logic network, making use of only the Boolean logical operators AND, OR and NOT. Meyers [32] describes a series of steps for determining cases using a cause-effect graph as follows:

- Divide the specification into workable pieces. A workable piece might be the specification for an individual transaction. This step is necessary because a cause-effect graph for a whole system would be too unwieldy for practical use.
- Identify causes and effects. A cause is an input stimulus, e.g. a command typed in at a terminal, an effect is an output response.
- Construct a graph to link the cause and effects in a way that represents the semantics of the specification. This is the cause-effect graph.
- Annotate the graph to show impossible effects and impossible combinations of causes.
- Convert the graph into a limited-entry decision table. In this case, conditions represent the causes; actions represent the effects and rules (columns) represent the test cases.

The purpose of the cause-effect graph is to identify a small number of useful test cases.

3.3 Structural Testing

The opposite to the black box approach is the *white box* approach. Here testing is based upon the detailed design rather than on the functions required of the program, hence the name structural testing. Structural testing is concerned with testing its implementation. Although used primarily during the coding phase, structural testing should be used in all phases of the life cycle where the software is represented formally in some algorithmic, design, or requirements language.

There are three kinds of structural testing methods based on the process of generating test data:

- *Statement testing*: the design of test data in order that all statement in the program should be executed at least once.
- *Branch testing*: enough test data to be written so that all branches in the program should be executed at least once.
- *LCSAJs testing*: enough test data required to be written so that all linear code sequence and jumps (LCSAJs) should be executed at least once.

Computation testing is another form of structural testing. This uses the structure of the program and select paths which are used to identify domains. The assignment statements on the paths are used to consider the computations on the path. These approaches also make use of the ideas of symbolic execution.

Computation testing strategies focus on the detection of computation errors. Test data for which the path is sensitive to computation errors are selected by analysing the symbolic representation of the path [11].

There are other software testing techniques. They are either functional testing or structural testing depending on the generation of test data. They may be thought as another kind of testing techniques. In some of these approaches, the input space of a program is partitioned into path domains, i.e. subsets of the program input domain that cause execution of each path and the program is executed on test cases which are constructed by picking test data from these domain. Examples of such techniques are *symbolic testing*, *algebraic program testing*, *grammar-based testing* and *data-flow guide testing*. Another approach is to instrument the program by recording processes which do not affect the functional behaviour, but record properties of the executing program.

There are another two important testing techniques: *domain testing* and *mutation testing.*

In domain testing there are two methods to select test data. In the first method, test cases are created based on informal classification of the requirements into domain; either data or function may provide the basis for the domain partitioning. The test cases are executed and compared against the expectation to determine whether faults have been detected. In the second method, test cases are created based on the observation that points close to, yet satisfying boundary conditions are most sensitive to domain errors[11]. An error in the border operator occurs when an incorrect relation operator is used in the corresponding predicate, and an error in the position of the border occurs when one or more incorrect coefficients are computed for the predicate interpretation. The domain testing strategy selects test data on and near the boundaries of each path domain.

Mutation testing is not concerned with creating test data and demonstrating that the program is correct. It is concerned with the quality of a set of test data[8] [7]. While other forms of testing use the test data to test the program, mutation testing uses the program to test the test data.

High quality test data will exercise a program thoroughly. To provide a measure of how well the program has been exercised, mutation testing creates many, almost identical, programs. One change is made per mutant program. Each mutant program and the original program are then executed with the same set of test data. The output from the original program is then compared with the output from each mutant program in turn. If the outputs are different then that particular mutant is of little interest as the test data has discovered that there is difference between the programs. This mutant is now dead and disregarded. A mutant which produces output that matches with the original is interesting. The change has not been detected by the test data, and the mutant is said to be alive.

Once the output from all the mutants has been examined, a ratio of dead to live mutants will be available. A high proportion of live mutants indicates a poor set of test data. A future set of test data must be devised and the process repeated until the number of live mutants is small, indicating that the program has been well tested.

Mutation testing is highly promising technique for judging the effectiveness of test data. In some cases it can be used to prove the correctness of certain types of programs. The major stumbling block is the problem of equivalent mutants. For example, if a PASCAL program contained the statements:

```
REPEAT
read (X)
S := S + X
UNTIL S > P
```

and the program was mutated to contain the fragment

```
REPEAT
read (X)
S := S + X
UNTIL S > Q
```

and, if in the program p and q were set to the same value then the mutant and the original program are equivalent. It requires the tester to examine the original and the mutant to check for equivalence. No test data would be able to distinguish between them. If a large number of equivalent mutants are created a considerable amount of work will be needed.

3.4 Static Analysis

A testing technique that does not involve the execution of the software with data is known as *static analysis.* In static analysis, the requirements and design documents and the code are analysed, either manually or automatically, without actually executing the code. Common static analysis techniques include such compiler tasks as syntax and type checking. Only limited analysis of programs containing array references, pointer variables, and other dynamic constructs is possible using these techniques. The static analysis techniques include requirement analysis, design analysis, code inspections, proof of correctness and walkthroughs.

3.5 Dynamic Analysis

Dynamic analysis requires that the program testing be executed, and hence follows the traditional pattern of program testing, in which the program is run on some test cases and the results of the program's performance are examined to check whether the program has operated as expected.

Functional testing may dictate the set of test cases. The execution of these test cases may then be monitored by dynamic analysis. The program can also be examined structurally to determine test cases which will exercise the code left idle by the previous test. This dual approaches results in the program being tested for the function required and the whole of the program being exercised. The latter feature ensures that the program does not perform any function that is not required.

3.6 Testing Tools

Testing is an important and budget-consuming part of the program development cycle. The manual testing process is especially tedious and error-prone. All or part of this process should be automated in order to reduce the number of errors. Automated testing tools should be able to analyse a program, delete some type of errors, generate test case, initiate program execution, log test results, and compare test results with expected results.

Testing tools are software tools that assist the testing of programs, such as analysing program structure, generating test data and recording test execution. They should be more cost effective to obtain a general program which can perform its function in a variety of test situations. The tools cover a wide range of activities and are applicable for use in all phases of the software development life cycle. Some perform static testing and others dynamic; while some evaluate the system structure, and others the system function.

A tool is a vehicle for performing a test process. The tool is a resource to the tester, but by itself is insufficient to conduct testing. A testing technique is a process for ensuring that some aspect of an application system functions properly. There are few techniques, but many tools. The concept of tools and techniques is important in the software testing process. It is a combination of the two that enable the test process to be performed. We should first understand the testing techniques and then understand the tools that can be used with each of the techniques.

3.6.1 Static Analysis Tools

A *static testing tool* (static analyser) is a program that analyses source code to reveal errors or dangerous constructs without actually executing the code. A static analyser is mainly used to check certain global aspects of program logic, syntactic errors, coding styles, and interface consistency. The information revealed by static analysers include:

- Syntactic error messages,
- Number of occurrences of source statements by type,
- Cross-reference maps of identifier usages,
- Analysis of how the identifiers are used in each statement (data source, data sink, calling parameter, dummy parameter, subscript, etc),
- Subroutines and functions call by each routine,
- Uninitialised variables,
- Variables set but not used,
- Isolated code segments that cannot be executed under any set of input data,
- Departures from coding standards (both language standards and local practice standards), and
- Misuse of global variables, common variables, and parameter lists (incorrect number of parameter).

There are many kinds of static testing tools which have been used, for example: *Front End Language Processor; Acceptance Test Criteria; Data Flow Analyser; Control Flow Analyser; Error Analyser; Report Generator, etc.*

Because the exact value of the variable may not be known until the execution time, one cannot generally know which array element is being referenced or defined in a statement. Therefore all static analysis tools are limited by the problem of identification when the subscript is a variable.

3.6.2 Dynamic Analysis Tools

Dynamic analysis tools provide support for testing by direct execution of the program being tested. The range of functions supported by a dynamic tool is broad. Systems, which generate and evaluate test data using any one or a combination of the testing techniques have been implemented and used in a variety of setting. Dynamic analyser tools can be divided into four parts: symbolic evaluators, test data generators, program instrumenters, and program mutation analysers.

1. *Symbolic Evaluators* — Symbolic evaluators are programs that accept symbolic value for some of the inputs and algebraically manipulate these symbols. They perform operations symbolically as if the program were executing and derive output values as symbolic expressions including the input variables.

2. *Test Data Generator* — A test data generator is a tool which assists a user in the generation of test data for a program or module. The purpose is to relieve the effort required in generating a large volume of test data, and in the case of automatic test data generation, to avoid programmer's bias in preparing his own test data. The same way that manual testing

can be regarded as two distinct approaches — functional and structural — automated generation of test data may be categorised into three types:

(a) Pathwise test data generators

(b) Data specification systems

(c) Random test data generators.

3. *Program Instrumenters* — Program instrumenters are systems that insert software probes into source code in order to reveal their internal behaviour and performance. Their main applications are coverage analysis, assertion checking, and detection of data flow anomalies. There are three types of program instrumenters: (1) dynamic execution verifiers, (2) self-metric instrumenters and (3) dynamic assertion processors.

4. *Mutation Testing Tools* — An automatic mutation system is a test entry, execution, and data evaluation system that evaluates the quality of test data based on the results of program mutation. In addition to a mutation "source" that indicates the adequacy of the test data, a mutation system provides an interactive test environment and reporting and debugging operations which are useful for locating and removing errors.

3.7 Summary

Testing is necessary at all stages of the software life cycle. Testing is an activity that requires a great deal of planning. It uses information from all previous phases. Therefore the test plan is a crucial document in the software development life cycle. It is very important to plan for proper testing, to work with objective testing criteria, to combine functional and structural testing, and to control the testing process as any other phase in the development life cycle.

Structural and functional testing are not competing approaches. Both methods are essential. Both are effective, and both have limitations. The two concepts actually represent the extreme points of a spectrum between structure and function. Unit testing tend to be more structural than functional, while the converse is true for system testing. Functional testing can in principle detect all bugs, but would take an infinite amount of time to do so. Structure testing cannot detect all bugs even if completely executed.

Strategically speaking, it is important to develop a reliable and effective functional test approach because, first, functional testing can address errors of emission whereas most other test approaches cannot, and secondly, because bought-in software parameterised packages and some "generated" software cannot really be tested by other means [1]. Howden suggests functional testing be taken further [28].

The requirement for a test oracle derived from the above survey on software testing is useful in the development of the testing approach. A program can only be tested properly if the tester has the exact knowledge about what the program under test should and should not do. This justifies the requirement for a test oracle for all test cases. Such information, for deciding if a program is behaving correctly, can generally be derived from the specification of the program.

Chapter 4

Testing Oracles

4.1 Introduction

Software testing is an important stage in the software life cycle. Once the software is implemented in machine-executable form, it must be tested to uncover defects in function, in logic and in implementation.

Almost all of the research on software testing focuses on the development and analysis of theory of input data selection criteria and particular criteria such as path testing, data flow testing functional testing, and random testing. Methods have been developed for the automatic generation of input data satisfying these criteria. Many schemes which propose large quantities of test input data, do not address the problem of getting expected outputs to check the resulting outputs for correctness. In particular there is an underlying assumption that once the phase of selecting test data is completed, the remaining tasks are straightforward. But it is frequently more difficult to obtain the expected output. Consequently, *ad hoc* methods of getting the expected output often must be used including hand calculation, simulation and alteration. Researchers have been searching for mechanisms that can derive the expected output of each input of a program, which is usually known as the oracle problem.

Oracles are an external source or a process of examination which, for any given input description, can provide a complete description of the corresponding output behaviour (i.e., expected outputs) and determine whether the test output confirms with the expected output. The oracle may be a program, a program specification, a table of examples, a body of data that specifies the expected output of a set of test data as applied to a tested program, or simply the programmer's knowledge of how a program should operate. The oracle may take a variety of forms, including:

1. Manual examination of each output from each test run — this is the form of oracle that is widely used. Generally, the tester serves as this kind of oracle. The expected outputs are computed by hand or derived from a specification. Actual outputs are compared visually. It is commonly called the human oracle.

2. Manual generation of each output once, to be compared automatically with the output from each test run — this kind of oracle is generally used in dual coding (Two programs are produced using the same specification. The definition will be given in the next section). In dual coding technique, the comparison of the outputs of two programs must be done accurately and efficiently, since there are a lot of outputs from two programs. A monitor is needed to run the two versions of the programs, and compare their outputs.

3. Programs which generate < x, y> pairs, where y is the correct output from input x — some test case generators can get test input and expected output pairs in terms of the testing technique. The generator can serve as this form of oracle.

4. Programs which generate the correct output y for any input x — this is an ideal oracle. In this kind of oracle, expected output can be generated for any test input.

5. Programs which determine the correctness of any input/output pair < x, y> — this is a testing tool, called a comparator. It is used to compare and check actual output with expected output.

The characteristics of the available oracles have a dominating influence on the cost and quality of software testing.

4.2 Pseudo-oracle Approaches

A *pseudo-oracle* is an independently produced program intended to fulfill the same specification as the original program [13]. This technique is frequently called dual coding, and has been historically used only for highly critical software. The two programs which are to be produced in parallel by totally independent programming teams, are run on identical sets of input selection which must satisfy some pre-determined test data adequacy criterion. If the outputs are the same (or acceptably close in the case of numerical programs), the original program is considered to be validated. If, on the other hand, the outputs of the two programs do not agree, the two programs are examined using standard debugging techniques. The process is repeated until all discrepancies are resolved. [13] describes a technique for producing pseudo-oracles. Such a technique has been used in producing ultra-reliable software system. This technique has been used to build a system of automatic program testing in [3,27,35,37].

Since the research into the pseudo-oracles involves the comparison of the outputs of two programs, it must be possible to do this accurately and efficiently.

The comparison might be done manually, although frequently a monitor which runs the two versions of the programs, and compares their outputs will be also necessary.

Another method uses very high level programming language as a pseudo-oracle. This method is similar as the dual programming method. The functional specification is given to two programmers and each of whom independently code a separate program using a different programming language. There are some languages by which it is relatively easy to construct program quickly. They can offer as much as a ten to one reduction in lines of code written as compared with conventional programming languages, e.g., building a high level language for testing on assembly language program. Typical very high level languages are: SETL, which is based on the storage of data in mathematical objects known as sets [39]; APL, which is based on the storage of data in arrays and PROLOG, which is based on logic. Girard and Rault described a project in [22] which used APL as a medium for producing oracles.

Although very high level languages are able to offer large reductions in programming time they do suffer from the disadvantages that the programs constructed tend to be inefficient. However, these programs can be used as oracles during the development of the more efficient production software.

The methods for constructing a pseudo-oracle are expensive, but it might be necessary for non-testable programs [46]. The following classes of programs are identified as being non-testable programs according to the definition given:

1. Programs which were written in order to determine the answer in the first place. There would be no need to write such programs, if the correct answers were known.

2. Programs which produce so much output that it is impractical to verify all

of it.

3. Programs for which the tester has a misconception. This may be thought of as a case in where there are two distinct specifications. The tester is comparing the outputs against a specification which is different from the original (given) specification.

For those programs deemed non-testable due to a lack of knowledge of the correct answer in general, there are nonetheless, frequently simple cases for which the exact correct result is known. In the case of programs which produce excessive amounts of output, testing on simplified data might involve minor modifications of the program. The problem with relying upon results obtained by testing only on simple cases is obvious. It is frequently the 'complicated' cases that are most error-prone. It is common for central cases to work perfectly where boundary cases cause errors. A pseudo-oracle may be necessary for these programs.

In order that a pseudo-oracle be useful in practical contexts, certain assumptions must be fulfilled [13]:

1. Independence of the pseudo-oracle

   This assumption is really central to our proposed methodology. The two (or several) programs must be developed completely independently by different programming teams. This is essential in order to eliminate the possibility of the some programmer's misconceptions being inserted into both the original program and the pseudo-oracle.

2. Availability of a convenient very high level language

   Obviously the team charged with the development of the pseudo-oracle must have an available compiler for a language in which code can only be written quickly and easily. The compiler should have substantial debugging features in order to facilitate the development.

3. Extensive use envisioned for original program

   The overhead involved in producing a second program (even in a very high level language) can only be justified if the original program is intended to be run often or in a safety critical environment.

4. Complete and precise specification

   There must be a complete and precise specification available to both programming teams. This is obviously crucial; it is hardly to be expected that two (or more) programs written to meet some vague incomplete specification will turn out to be equivalent.

4.3 Attributed Grammar Approaches

This method for producing oracle uses an *attributed context-free grammar* as the basic mechanism for describing and generating test inputs and expected outputs. The attributes provide the context sensitive information necessary to generate semantically meaningful test cases.

4.3.1 Context-free Attributed Grammars

A context-free grammar is important concept for constituting an oracle in this thesis. This concept is defined here and will be used in the next chapters.

Sentence

An *alphabet* is a collection of characters. The ASCII character set is a good example of an alphabet. A *string* or *word* (*token*) is a specific sequence of symbols from an input *alphabet.* A *language* is a set of words, and a *sentence* is a sequence of one or more of the words within the sentence, and that is where a *grammar* comes into play. A formal *grammar* is a system of rules (called *productions*) in which the order of words may occur in a sentence.

The *syntax* of a sentence determines the relationships between the words and phrases in a sentence. That is, the syntax of a language controls the structure of a sentence.

Context-free Grammar

A *context-free grammar* is a system of definitions that can be used to break up a sentence into phrases solely on the basis of the sequence of strings in the input sentence. A context-free grammar is usually represented in Backus-Naur form (BNF). This notation has a number of significant advantages as a method of specification for the syntax of a language[2].

1. A grammar gives a precise, yet easy to understand, syntactic specification for the programs of a particular programming language.

2. An efficient parser can be constructed automatically from a properly designed grammar. Certain parser construction processes can reveal syntactic ambiguities and other difficult-to-parse constructs which might otherwise go undetected in the initial design phase of a language and its compiler.

3. A grammar imparts a structure to a program that is useful for its translation into object code and for the detection of errors.

In general, BNF involves four quantities: terminal, non-terminal, a start symbol, and productions. The basic symbols of which strings in the language are composed we shall call terminals. The word "token" is a synonym for "terminal" when we are talking about programming languages. Non-terminals are special symbols that denote sets of strings. The terms syntactic 'variable' and syntactic categories 'statement', 'expression', and 'statement-list' are non-terminals; each denotes a set of strings. One non-terminal is selected as the start symbol, and it denotes the language in which we are truly interested. The other non-terminals are used to define other sets of strings, and these help to define the language, they also help to provide a hierarchical structure for the language at hand.

The productions define the ways in which the syntactic categories may be built up from one another and from the terminals. Each production consists of a non-terminal, followed by the symbol "::=", followed by a string of non-terminals and terminals.

For example, consider the following grammar for simple arithmetic expressions. The non-terminal symbols are 'expression' and 'operator', the 'expression' being the start symbol. The terminal symbols are

ID + - * / ( ).

The productions are:

expression ::= expression operator expression

expression ::= (expression)

expression ::= -expression

expression ::= ID

operator ::= +

operator ::= -

operator ::= *

operator ::= /.

Attribute Grammar

An *attribute grammar* provides a formal method for specifying the semantics of sentences in a language which is defined by context-free grammar [24]. An attribute grammar consists of a set of context-free productions each of which has an associated set of rules expressed in the form of semantic functions. The attribute grammar associates a finite set of attributes with each grammar symbol. The semantic functions specify the way in which the attributes of particular symbols are to be evaluated from the attributes of other symbols in the same production.

An attribute grammar gives a theoretical basis for the computation of semantic attributes, and assist in the semi-automatic production of semantic analysers. In time, the automatic production of semantic analysers from attribute grammars may become as commonplace as the production of table-driven parsers from context-free grammars is at present.

The attribute grammar specifies the semantic attributes of any sentence in terms of purely local properties. Attribute values only depend on the attributes of neighbouring nodes in the tree. In effect, the semantic properties of a language are reduced to stepwise computations in exactly the same way as the productions of a context-free grammar reduce the syntax of a language to computations which are performed one production at a time.

4.3.2 Attributed Grammar Oracles

An example of BNF is shown below:

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

< integer > ::= < digit > | < digit > < digit >.

This shows the rules forming an integer in a programming language. The first rule states that a digit is either a zero, a one, a two, etc. The second rule states that an integer can either be a digit or a digit followed by an integer. Thus 372 is an integer. It consists of a 2 which is a digit and hence an integer. The integer 2 is preceded by the digit 7 which makes 72 an integer. The integer 72 is preceded by the digit 3 which makes 372 an integer. A compiler for a programming language can use such rules in order to process programs.

The method for producing oracles using attributed grammar reverses this process. A tester defines a test grammar. The basic test grammar is written using an extended BNF. The generation process for a grammar starts with an empty string and then generates an example of start symbol. It does this by choosing one of the rules for the start symbol. The choice procedure is described as follow. Once a group of symbols occurring to the left of the predicate in the rule. The guard may also involve the value of the current terminal or any of its synthesised attributes. If the guard has a true value then the rest of the term is evaluated, otherwise it is not and the next alternative term, if any, is processed.

As with compilers, action routines are used to produce an output string corresponding to the generated input string. Action routines are allowed any place in the right-hand-side of the rules, and these action routines and their attributes specify those actions necessary to produce the output string.

Suppose to generate a test case involving N ($\leq$ N_MAX) integers from the range V_MIN to V_MAX. Basing attributed grammar method on the output, a sort list of numbers will be generated for the output file. The output string is presented by the context-free grammar:

sort_output ::= empty | sort_output "element" sort_output

The attributes will be used to ensure that the output string is given length and that the element are sorted. The following attributed grammar is defined.

ATTRIBUTED TEST GRAMMAR

1. test-case ::= [# N in 0..MAX_N]@init sort_output(N, V_MIN, V_MAX).

2. sort_output(N, A, B) ::= [? N= 0] empty | [? N > 0][# I in A..B; j in 0..N-1] sort_output(J, A, I) "element"(I) @put_element (I) sort_output(N - J - 1, I, B).

ACTION

init: initialises the input string to be empty.

put_element(I): puts the integer I into some random free location in the string.

Since the grammar productions are carried out in a top-down, left-to-right fashion, the grammar will build the output string from left to right. The two grammars above will work well enough with test data.

Several approaches for producing oracles involve the use of this technique. But more approaches focus on automated testing and test case generation. Generally, the approaches to oracle construction are tackled as part of the test generation.

Duncan and Hutchison presented this method for generating test cases in [15]. The user of a test case generator using attributed grammars defines the grammar of the test data to be generated, then takes this grammar and produces test data together with expected output.

Using this technique, Panzl [34] reported on regression testing of FORTRAN subroutines. He presented a test case description language and a program to automatically execute the cases, monitoring actual versus expected behaviours. Jagota and Rao [30] have developed a test case language and an interpreter, specifically designed for testing microprocessor operating systems. Camuffo et. al. [9] proposed the description of the functional specifications of a program (syntax and semantics) using a formalism based on two-level grammars. Some tools, which can generate automatically both test data and related expected output are implemented. Elsewhere, attributed grammars have been used both as test-case generators and test oracles.

The method for producing test cases using attributed test grammar is quite expressive, but is not very user-friendly in appearance. One disadvantage of this method is that the generated test case, while syntactically correct, can be semantically wrong [29]. Another is that the tester needs to write the attributed test grammar. As yet there has been very little experience in the use of such grammars. For example, it is still not yet clear whether for production quality programs the complexity of the grammar produced will be so large as to make the technique unfeasible.

4.4 Formal Specification Approaches

4.4.1 Formal Specification Languages

A formal specification will describe the following:

- what input the system is going to process,
- what function it will perform for each input, and
- what the corresponding output will be.

Any specification must be in a specification language. Specification languages may be classified into two major classes: formal specification languages and informal specification languages .

Formal specifications have a mathematical (usually formal logic) basis and employ a formal notation to model system requirements.

Informal specification languages, on the other hand, use a combination of graphics and semiformal textual grammars to describe and specify system requirements. Given the graphical and "English-like" nature of these languages, they provide a vehicle for eliciting user requirements and communicating the analyst's understanding of the requirements back to the user for verification.

The two approaches have complementary strengths and weaknesses. Whereas informal specifications have advantages for requirements elicitation, ease of learning, and communication, formal languages provide conciseness, clarity, and precision, and are more suitable for analysis and verification. Therefore, formal and informal specifications must not be regarded as competitive but rather as complementary.

4.4.2 Oracles Based on Formal Specification

Specifications are of great importance in testing, for they determine what the software ought to do and must necessarily form the basis for the testing of the functionality of software.

The formal specification language has two major components: syntactic specifications and semantic specifications. The syntactic specifications provide syntactic and type-checking information, and range of operations. Semantic specifications define the meaning of the operations by starting, in the form of axioms, the relationships of the operations with each other. The formal functional specifications describe the program behaviour, i.e., syntax part describes all the legal input to the program, together with the 'meaning' of each given by the semantic part.

For formal specifications, we can use formal specifications as an oracle in one of two ways:

o to check the result, and

o to predict the result.

Checking the result seems in principle easier. The alternative would seem to use the specification to calculate or predict the output that should be expected, but that requires an executable specification, and many specifications will be essentially non-executable. There are some approaches for producing oracles using formal specifications, typically the oracle problem is tackled as part of the test case generation activity.

Day and Gannon implemented a test oracle based on formal specification [14]. This system translates a formal specification of input-output data into an automated oracle. The generated oracle will validate the consistency of an unlimited number of program inputs and outputs with the specification. The specification language used in the system is intended to describe a problem domain limited to programs written in CF PASCAL (Character File Pascal). CF PASCAL is a Pascal subset which allows the programmer only the two primitive data type, CHAR and TEXT, with all program inputs and outputs in the form of text files. Strictly speaking, CF PASCAL is not a formal specification language. The specification used does not give enough semantics attributes so that a user must define functions to specify other semantic attributes. The users must understand attribute evaluation and bottom-up parsing, and attribute-evaluation software would have to be written. The following example shows that Day and Gannon gained a test oracle based on their own formal specification.

This example is a specification for a program which is to remove extra blanks which separate the words in the input file.

PROBLEM: Write a program to take as input a line of text and output the text with a single blank between each word. All initial blanks should also removed form the line.

SYNTAX SECTION

```
<filein>::= <wblist> <eoln>
          | <blank> <wblist> <eoln>
          | <wblist> <blanks> <eoln>
          | <blanks> <wblist> <blanks> <eoln>
          | <blanks> <eoln>
          | <eoln>
<wblist> ::= <wblist> <blanks> <word> | <word>
<word> ::= <char> <word> | <char>
<blanks> ::= <blank> <blanks> | <blank>
<fileout>::= <wlist> <blanks> <eoln> | <blanks> <eoln>
<wlist> ::= <wlist> <blank> <word> | <word>
<word> ::= <char> <word> | <char>
<blanks> ::= <blank> <blanks> | <lambda>
```

The addition of the semantic rule finishs the section.

SEMANTIC SECTION

List(Words(Fileout)) = List(Words(Filein))

An execution of the generated oracle for the insert blanks problem is showed below. Note that the output file is incorrect; the extra blanks were all inserted after the first word.

```
Width :                                     /
Filein  : List of words separated by one blank/
Fileout: List of words separated by one blank/
Oracle Run: $iblanks
Checking Syntax Rule 1.
```

```
Data Meets Specification SYNTAX RULE 1.
Checking Syntax Rule 2.
Data Meets Specification SYNTAX RULE 2.
Checking Syntax Rule 3.
Data Meets Specification SYNTAX RULE 3.

Checking Semantic Rule 1.
Data Meets Specification SEMANTIC RULE 1.
Checking Semantic Rule 2.
Data Meets Specification SEMANTIC RULE 2.
Checking Semantic Rule 3.
Data Meets Specification SEMANTIC RULE 3.
Checking Semantic Rule 4.
Semantic Error: SEMANTIC RULE 4.
```

There are other approaches for producing an oracle based on formal specification language as part of test case generation.

Frankl [18] has developed a schema for object-oriented testing using algebraic specifications to test an object O. The specification is used to generate pairs of equivalent call sequences to instances of O and one of the call sequences is executed on each instance. Object behaviour is deemed correct if the two instances are left in the same internal state. With the Mockingbird system [23], the specification for a message protocol is transformed to a constraint logic program which can serve as a validator and generator of test data. As a generator, the program produces messages conforming to the specification, as well as syntactically and semantically incorrect messages. As a validator the program serves as the test oracle, Choquet

[10] and Pesch [36] used specifications to generate test input and expected output. To obtain the test cases from these specifications particular values are substituted for variable in parts of the specification, it is the tester's responsibility to generate the appropriate instantiations. Gerhart [19] describes an interactive system for generating test data using Prolog. The user of the system identifies important conditions in the specification of a program and provide a generator for test data. The system then generates a set of test data that covers as many combinations of the conditions as possible. Tsai [44] described a system of automated test case generation for programs specified by relational algebra (RA) queries. To automate test case generation, limiting the expressive power of the specification language (relational algebra RA) is used.

At present, creating test cases from formal specifications is a complicated and poorly understood information-processing technique requiring extensive human expertise. Thus the specification language which is used for producing oracles has to be limited by the expressive power.

4.5 A Summary of the Approach to the Construction of Oracles

In general, we can classify oracles into four kinds in terms of their construction: human oracles, pseudo-oracles and oracles using attributed grammars and specification-based oracles. For human oracles, one method is that expected output is derived manually from specification, but this approach can be quite expensive especially when a large mass of test cases must be executed, as is normally the case, for example, with random testing; another method is that expected output is not derived at all and the effective actual output is inspected by the operator:

this may be a dangerous testing practice since the testing evaluation becomes dependent on the competence and the fairness of the operator [6]. Even so, human oracles are frequently used in software testing.

The use of a pseudo-oracle for testing may not be practical. Obviously such a treatment requires a great deal of overhead. At least two programs must be written, and if the output comparison is to be done automatically three programs are required to produce what one hopes will give identical result. It has been argued [20] that writing multiple versions of software, even when the language used is not a very high level language, does not add substantially to the cost of a software system. Two programs developed in similar ways may contain similar errors. In addition, the construction of multiple versions of software as oracle reduces the large amount of effort expended during testing.

With the development and application of a formal specification language, the approach of using specifications as an oracle has received a lot of attention. But these approaches have not been sufficiently well-defined to be generally applicable [38]. There has been very little experience of using attribute grammars. Besides, the specifications which are used in the above approaches are test specifications. These specification languages are only used to ease software testing and they may not be useful for other purposes. Thus, the extension of these methods will be restricted, and an extra cost will be paid for writing extra testing specification.

Chapter 5

The Construction of an Automated Oracle

5.1 A Scheme for an Automated Oracle

Specifications are of great importance in testing, for they determine what the software ought to do and must necessarily form the basis for the verification testing of the functionality of the software [25]. The use of a formal specification allows the development phase and test preparation to be performed due to the proper advantages of formal specification. For a formal specification, it must have a mathematical (usually formal logic) basis and employs a formal notation to model a system. The advantages of using formal specification [40] are as follows:

- The development of a formal specification provides insights into and understanding of the software requirements and the software design.
- Given a formal system specification and a complete formal programming language definition, it may be possible to prove that a program conforms to its specification. Thus, the absence of certain classes of system error may

be demonstrated.

- Formal specifications may be automatically processed. Software tools can be built to assist their development, understanding and debugging.
- Depending on the formal specification language used, it may be possible to animate a formal system specification to provide a prototype system.
- Formal software specifications are mathematical entities and may be studied and analysed using mathematical methods.
- Formal specifications may be used as a guide to the tester in identifying appropriate test cases.

We can consider making use of formal specifications to construct an automatic oracle for checking or predicting the expected output of a software system.

A formal specification language is needed to describe both the syntactical (also for contextual parts) and semantic aspects of the software. The contextual part of a specification describes all the legal input to the program, the semantics part describes the meaning (expected output) of each given input data. In terms of this idea, an oracle can be constructed from a context-free grammar, together with the related "meaning". Figure 5.1 shows a schema of the technique.

A *scanner* is a program which performs lexical analysis of source language, as opposed to syntactic analysis and semantic analysis. Lexical analysis is that part of the compiler which reads the original source language character by character and translates it into a sequence of primitive units called tokens or terminal symbols. The function of a scanner is to take an input source language and produces as output a stream of tokens suitable for the syntax analyser or parser. There are only a small number of tokens for any language and they are conveniently represented

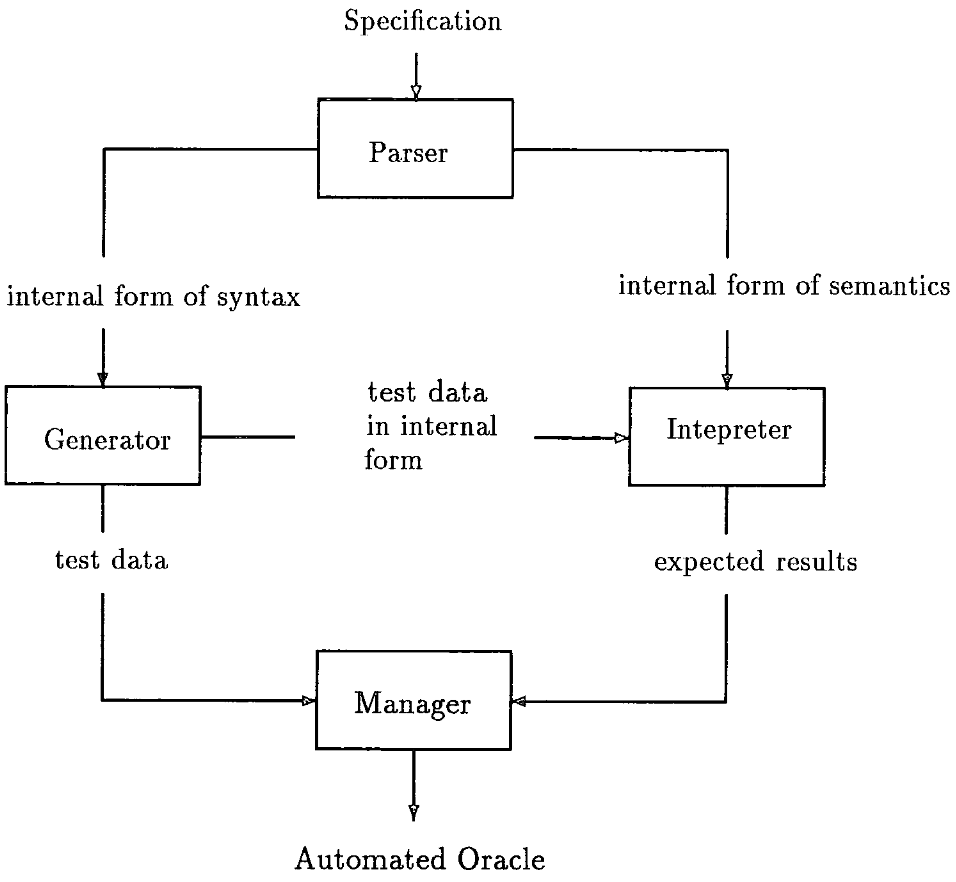

Figure 5.1: A Scheme of Automatic Oracle

by small integers. The scanner can recognise symbols normally defined by a context-free grammar.

A scanner is often sufficiently simple that it can be written for example straight as a straightforward C program. Such a scanner (lexical analyser) consists of a loop containing a single switch statement which decides what the next token being read is by looking at the next character. It takes as input the source text and produces as output a list of tokens. There are suitable tools for the construction of a lexical analyser, of which Lex [5] is one of them. Lex automaticly generates lexical analysers and it will be described in the next section.

A *parser* is a program which performs syntax analysis. A parser determines whether the stream of tokens from the scanner forms a valid sentence in the source language grammar. If so, a parse tree can be unambiguously derived.

There are two obvious ways of building up a parse tree. One is to start with the sentence symbol and build downwards to the terminals; the other is to start at the terminal and build upwards to the sentence symbol. These are known as top-down and bottom-up parser methods respectively.

In the top-down parsing, the LL(1) parsing method is widely used. The LL(1) parser means a parser in which we read the source text from the left (the L), and then produce a left-most (the L) derivation. We recognise non-terminal in turn, starting with sentence symbols to form the parse tree.

In the bottom-up parsing, the LR(k) shift-reduce parser is used. There are three types of LR parser: canonical LR (k), simple SLR(k) and lookahead LALR(k). In general, we are interested in the case where k=1.

With shift-reduce parsing, we use a stack to hold symbols. At any stage, we have principally two options:

- Push the current token onto the top of the stack and call the lexical analyser to get a new token. The token is said to be shifted onto the stack.
- Decide for the token on the top of the stack from a valid right-hand side of a production. Pop them off the stack and replace with the non-terminal on the left-hand side of the production. This is known as reduce using production.

Some parser generators are widely used now. A parser generator is a complete programming language which can generate automatically a parse table. YACC [5] is an outstanding one. YACC is widely available under Unix and some other operating systems. YACC takes a specification of a programming language grammar and semantic actions and produces an LALR(1) parsing table and a shift-reduce parser. The source program is read as a stream of tokens. A lexical analyser must be provided separately, typically using the Lex lexical analyser generator.

A *test data generator* is a program that can generate syntactically and contextually correct test data (input data) through a right-most derivation of the attribute syntax given in the specification. Given the syntax and a representation for the test-domains, the algorithm for generating the test domain partition is produced for a particular specification. Typical test data are selected in terms of the given test domain partition. The work of this part focuses on how to generate test input data. In this thesis, we only pay our attention to how to get expected output from the input data which has been given.

An *interpreter* is a program that simulates the behaviour of the software under validation, by 'execution' of the semantics, and produces the expected results relating to the test domain. It needs to give special semantics to a formal specification language and then interpret the formal specification language using this semantics. YACC provides a function for semantic action. The user may specify actions that are executed whenever a rule (production) or part of rules, is recognised. These actions can return values and access values returned by previous actions. These features of YACC can also be used to write an interpreter in C.

A *manager* is a program that aggregates the generated test data and expected outputs, and prints an testing oracle in the form of a case table.

According to this idea, an expected output related test-domain can be obtained from a formal specification. The pre-requisite would be the description of syntax of the specification language. The Z specification language has concrete syntax and semantics, and is almost fully defined [41,42] so it is used in this thesis.

5.2 Z Specification Language

5.2.1 Features of Z Specification Language

There are two approaches to formal specification languages which fundamentally affect the way in which a system is specified, the model-oriented and algebraic approach:

- A model-oriented specification aims to construct an explicit

  abstract model of an information system in terms of well-understood mathematical entities such as sets whose semantics are formally defined.

- An algebraic approach specification involves creating objects which represent some real world entities, and model them in terms of the operations which can be performed on them.

Z is a model-based specification language. Z specifications are more intuitive to non-scientists since it models real-world entities directly using relatively simple mathematical objects.

Z is a formal specification language devised by Jean-Raymond Abrial and developed by the Programming Research Group at Oxford University in the early 80s. It is still a topic of research at Oxford and other institutions, and has been the centre of interest to the non-academic world.

Z has been used to specify several non-trivial information systems. It is mainly used for safety-critical projects at present, but this situation will hopefully change as more and more people use Z.

Z is of interest to academics because of its mathematical foundations, and the promise of being able to bring a degree of rigour to a software engineering project. However, there is some doubt whether this can be achieved universally. This is probably right, but that does not detract from the very real benefits discussed in the previous section. These benefits could be secured by sing Z in the real world.

One of the biggest advantages of Z as a language for specifying medium to large-size systems is its in-built *schema calculus*. This provides a mechanism for easily decomposing specifications into smaller, more manageable units called *schemas*. A schema consists of a collection of named objects with a relationship specified by some axioms, and Z provides notions for them in various ways, so that a large specification can be built up in stages. Schemas can have generic parameters, and there are operations in Z for creating instances of generic schemas. Schemas give Z specifications a modular property, something which has long been recognised as a powerful aid when dealing with complicated problem domains. It helps the analyst to build a correct specification, and allows the reader to be gently introduced to a new specification by gradually unravelling the model. It basically reflects the human shortcoming of only being able to deal with a few new concepts at any one moment in time.

5.2.2 Examples of Z

This example specifies a simple banking system[41]. We begin by deciding that the state of the system consists of the balance of each account

$$\begin{array}{|l}
\textit{Bank} \\
\hline
bal : ACCT \mapsto \mathbb{N} \\
\hline
\end{array}$$

The arrow $\mapsto$ indicates a function from ACCT to N (natural number).

When the operation changes the states, we use Δ BANK to indicate it. When the operation does not change the states, we use Ξ BANK to indicate it. *bal* represents the state before an operation and *bal'* represents the state after an operation.

Δ BANK $\widehat{=}$ BANK $\wedge$ BANK'

Ξ BANK $\widehat{=}$ [Δ BANK | bal' = bal]

The balance is in number of pence. We suppose that the bank manager is mean enough to never allow overdrafts.

One possible operation is to transfer some money from one account *src* to another *dst*:

$$\begin{array}{|l}
\textit{Transfer1} \\
\hline
\Delta Bank \\
amount? : \mathbb{N} \\
src?, dst? : ACCT \\
\hline
src? \neq dst? \\
bal(src?) \geq amount? \\
bal' = bal \oplus \{src? \mapsto bal(src?) - amount?, \\
\qquad\qquad dst? \mapsto bal(dst?) + amount?\} \\
\hline
\end{array}$$

By convention names in the declarations ending in '?' are input data, and names ending in '!' are output data; the '?' '!' are otherwise just part of the name.

The operator $\oplus$ (function overriding) combines two functions of the same type to give a new function. The function f $\oplus$ g is defined as x if either f or g are defined, and will have a value of g(x) if g is defined at x; otherwise it will have a value of f(x). That is

$\mathrm{dom}(f \oplus g) = \mathrm{dom}(f) \bigcup \mathrm{dom}(g)$

$x \in \mathrm{dom}(g) \Rightarrow (f \oplus g)(x)=g(x)$

$x \notin \mathrm{dom}(g) \wedge x \in \mathrm{dom}(f) \Rightarrow (f \oplus g)(x)=f(x)$

We might describe also the operations of depositing and withdrawing money from the bank, asking for the current balance of an account, and so on, then turn later to the reporting of invalid operations. To do this, we add an extra output *report!* to each operation, and arrange such that this has a value 'OK' after every successful operation:

Ok

$report! : MESSAGE$

$report! = "ok"$

For unsuccessful operations, we report the reason for failure with an appropriate message, and constrain the final state of the banking system to be the same as the initial state. The two possible errors in a Transfer operation occur when the source and destination accounts are the same, and when the source account does not contain enough money:

SameAcct

$\Xi BANK$

$src?, det? : ACCT$

$report! : MESSAGE$

$src? = dst?$

$report! = \text{"}Same\ account\ for\ src\ and\ dst\text{"}$

NotEnough

$\Xi Bank$

$amount? : \mathbb{N}$

$src?, det? : ACCT$

$report! : MESSAGE$

$src? \neq dst?$

$bal(src?) < amount?$

$report! = \text{"}Not\ enough\ money\ in\ src\text{"}$

The transfer operation, complete with error reporting, can now be specified by combining these schemas:

Transfer $\widehat{=}$ (Transferl $\wedge$ OK) $\vee$ SameAcct $\vee$ NotEnough.

Transfer

$\Delta Bank$

$amount? : \mathbb{N}$

$src?, det? : ACCT$

$report! : MESSAGE$

$src? \neq dst?$

$bal(src?) \geq amount?$

$bal' = bal \oplus \{src? \mapsto bal(src?) - amount?,$

$\qquad dst? \mapsto bal(dst?) + amount?\}$

$report! : \text{"}OK\text{"}$

$\vee$

$src? = dst?$

$report! = \text{"}Same\ account\ for\ src\ and\ dst\text{"}$

$\vee$

$src? \neq dst?$

$bal(src?) < amount?$

$report! = \text{"}Not\ enough\ money\ in\ src\text{"}$

5.2.3 Identifying the test cases using Z

The most important consideration in program testing is the design or invention of effective test cases [32]. The typical test case design of functional testing techniques is **equivalence partitioning**. Equivalence partitioning is a technique for determining which classes of input data have common properties. The equivalence classes must be identified by using the program specification. There are two considerations for equivalence partitioning:

1. Each test case should invoke as many different input condition as possible in order to minimise the total number of test cases necessary.

2. One should try to partition the input domain of a program into a finite number of equivalence classes such that one can reasonably assume that a test of a representative value of each class is equivalent to a test of any other value. That is, if one test case in an equivalence class detect an error, all other test cases in the equivalence class would be expected to find the same error.

Using the equivalence partitioning techniques, we look for a partition of the input and output sets and states. These are given in the declaration parts of the specification, and the conditions contained in the predicates parts of the specification. In this particular case this can lead to the table below

bal	bal $\geq$ Amount		bal $<$ Amount
Amount?	$\mathbb{N}$	$\mathbb{N}$	$\mathbb{N}$
src? dst?	src? $\neq$ dst?	src? $=$ dst?	src? $\neq$ dst?
report!	OK	same account	not enough money
bal'	bal' $=$ bal $\oplus$ sth	bal' $=$ bal	bal' $=$bal

Note:

bal' $=$ bal $\oplus$ sth is in above table:

bal' $=$ bal $\oplus$ {src? $\mapsto$ bal (src? - amount?, dst? $\mapsto$ bal(dst?) $+$ amount? }

The report! and bal' are just expect outputs relating to inputs src?, dst?, amount? for the banking system. How to use this table will be discussed in the next chapter.

5.3 Lex

Lex is a lexical analysis generator.

The general format of Lex source is

{ definition }

%%

{ rules }

%%

{programmer subroutines}

The definition section is used to define variables for the program and for use by Lex. It may also contain other commands, including the section of a host language, a character set table, a list of start conditions, or adjustments of the default size of arrays within Lex itself for large source programs.

The rules section of a Lex input consists of a regular expression and an action. A regular expression specifies a set of strings to be matched. It also contains text characters (which match the corresponding characters in the strings being compared) and operator characters(which specify repetitions, choices, and other features). When an expression is matched, Lex executes the corresponding action, There is a default action, which consists of copying the input to the output. In Lex, the actions are written in C.

The third section is used for any subsidiary code that the user needs. This section can hold whatever auxiliary procedures are need by the action. Alternatively this part can be complied separately and loaded with the lexical analyser.

Lex accepts a high-level, program-oriented specification for character strings matching, and produces a program in a general purpose language which recognises regular expressions. The regular expressions are specified by the programmer in the source specifications given to Lex. The Lex written code recognises these expressions in an input stream and partitions the input stream into strings matching the expression. At the boundaries between strings, program sections provided by the programmer are executed. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

Lex does not worry about ambiguity. It will always select the longest possible match. If two matches are the same length, the first is used.

Lex is designed to simplify interfacing with YACC. What Lex writes is a program named yylex(), the name require by YACC for its analysis.

5.4 YACC–Yet Another Compiler-Complier

YACC is a parser generator which automatically generates LALR(1) parsing tables and a shift-reduce parser from a specification of the grammar and associated semantic actions. A YACC specification has three parts like

declaration

%%

rules

%%

programs.

The declaration part contains the declarations of all tokens that will be passed from the lexical analyser and used in the rules and programs sections.

The rule section is made up of one or more grammar rules and the associated semantic action. A grammar rule has the form

A : BODY ;

A represents a nonterminal name, and BODY represents a sequence of zero or more name and literals. The colon and semicolon are YACC punctuations.

Names may be of arbitrary length, and may be made up of letters, dot '.', and non-initial digits. Upper and lower case letters are distinct. The name used in the body of a grammar rule may represent tokens or nonterminal symbols. A literal consists of a character enclosed in single ' ' 's. As in C, the backslash '\' is an escape character within literals, and all the C escapes are recognised.

With each grammar rule the programmer may associate actions to be performed each time the rule is recognised in the input process. These actions may return values , and may obtain the values returned by previous actions. Moreover, the lexical analyser can return values for tokens, if desired. An action is an arbitrary C statement, and as such can have input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces '{' and '}'.

The third section of the YACC specification consists of supporting C routines to support the semantic actions defined in the rule section.

YACC provides a general tool for imposing structure on the input to a computer program. The YACC programmer prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognised, and a low-level routine to do the basic input.YACC then generates a function to control the input process. This function, called a parser, calls the programmer-supplied low-level input routine (the lexical analyser) to pick up the basic items (tokens) from input stream. These tokens are organised according to the input structure rules, called grammar rules; when one of these rules has been recognised, the programmer code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

Since YACC uses an LALR(1) parsing table, both shift-reduce and reduce-

reduce conflicts may occur. With shift-reduce conflicts YACC uses a shift in favour of a reduction. With reduce-reduce conflicts YACC uses the production declared first in the rules section. Both conflicts are permissible in programming language grammars.

A lexical analyser by the name yylex() must be provided. The lexical analyser yylex() returns tokens consisting of token type and attribute value pairs. If a token type value is returned as digit, the token type must be declared in the first section of the YACC specification. The attribute value is communicated to the parser by a YACC defined variable yyval.

The next chapter will discuss how to use LEX and YACC to implement an automatical oracle based on Z formal specification language.

Chapter 6

Implementation

The idea of constructing an automatic oracle is to make use of two features of the Z formal specification language — the concrete syntax and the semantics of the language. In this thesis, an automatic oracle generation consists of four parts. Figure 6.1 shows this system with four parts: a scanner, a parser, a syntax checker and an expected output generator. In this system, the input of the system is a particular specification written in Z; the output is the printed Z specification schema in boxes and a table of expected output related input-domain for this particular specification.

The scanner used for the lexical analysis is written in Lex. The parser is written in YACC. The parser reads a stream of tokens from the scanner and produces a symbol table in terms of Z grammars. The arrows between the scanner and the parser indicates the parser invoking the scanner and the lexical analyser returning the current token to the parser. The syntax checker utilises an existing tool — *fuzz*. *fuzz* is a package for checking Z specifications with the Z scope and types, and printing them with LaTeX. *fuzz* can be decomposed into two parts: one part is a program for analysing and checking specifications expressed as LaTeX input files; the other part is a LaTeXstyle option, containing environments for the

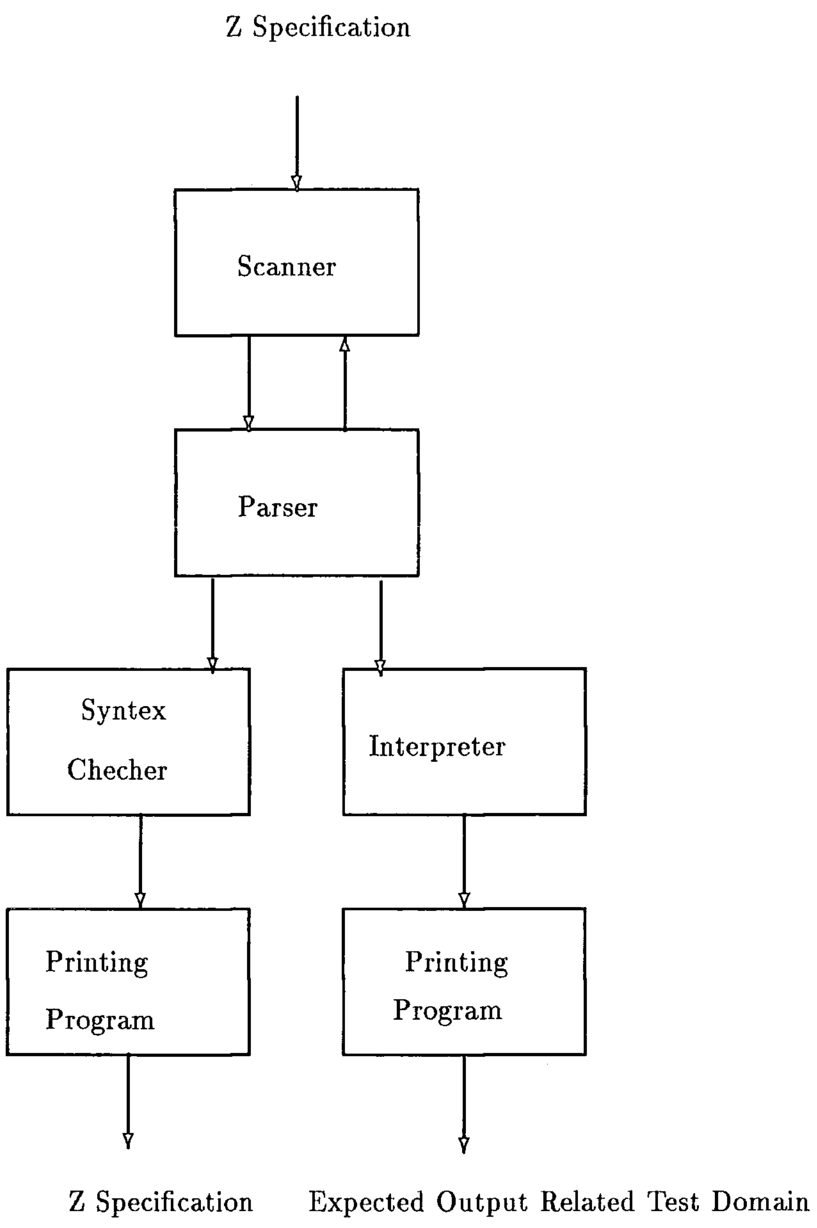

Figure 6.1: Oracle Generation

major constructs of Z and commands for the mathematical symbols.

The expected output generator (interpreter) is written in C. It consists of several semantic routines including action parts of YACC. In the semantic analysis, Z is interpreted using specific semantics of Z. The expected output generator produces a test case table and prints it.

6.1 Lexical Analysis and Scanner

Lexical analysis is the part of a complier which reads the original source program character by character and translates it into a sequence of primitive units called tokens or terminal symbols [17]. In this case, the original source program is a particular specification written in Z. Traditionally the program which performs this function is called a scanner. Thus a scanner performs a lexical analysis of the Z language.

The scanner is relatively straightforward to produce using Lex together with grammar and concrete syntax given in reference [42] and [43]. The diagram of the scanner is shown in Figure 6.2. The first phase predefines strings that will be used in phase two. For example, the Z lexical analyser includes the following definitions:

```
whitespace  {delimiter}+
alpha       [a-zA-Z_]
alphanum    [a-zA-Z_0-9]
digit       [0-9]
integer     {digit}+
```

Each definition consists of a name being defined on the left and its definition on the right.

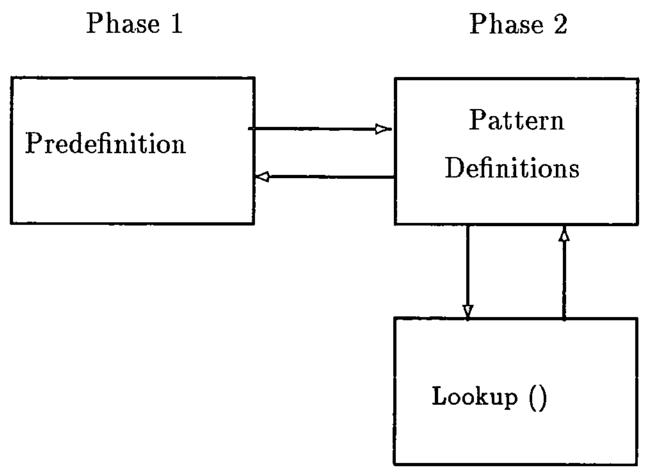

Figure 6.2: Lexical Analyser Generation

The second phase is where the patterns corresponding to each token defined in YACC are defined. When a pattern is matched, a corresponding action included in braces (some C code) is executed.

The subsidiary routine (*lookup()*) is then called, to recognise a token or a variable. When a variable is recognised, its name is saved in the symbol table. An external variable yylval is used to return a simple mapping for the value of each "integer". During the translation of the mathematical symbols into their ASCII representation, the LaTeXsymbols as tokens are used:

$\oplus$ to be translated to OPLUS

$\in$ to be translated to IN, etc.

6.2 Syntax Analysis and Parser

The purpose of syntax analysis is to determine whether the stream of tokens from the scanner forms a valid sentence in the Z specification language grammar. If so, its parse tree and symbol table is derived. The parser for Z is generated using YACC (See Appendix B). The diagram of parser is shown in Figure 6.3:

In phase 1, the tokens used in the grammar are declared. For example:

%token BEGIN_ZED

%token END_ZED etc.

The grammars given in [43] are ambiguous grammars. For the ambiguous grammars, YACC resolves the ambiguity by specifying the precedence and association of tokens, rather than merely rewriting unambiguous grammars. For the association, using "%left token-name" declaration for operators associated to the left; using "%right token" declaration for the operator associated to the right. Operators that have same the precedence appear in the same declaration.

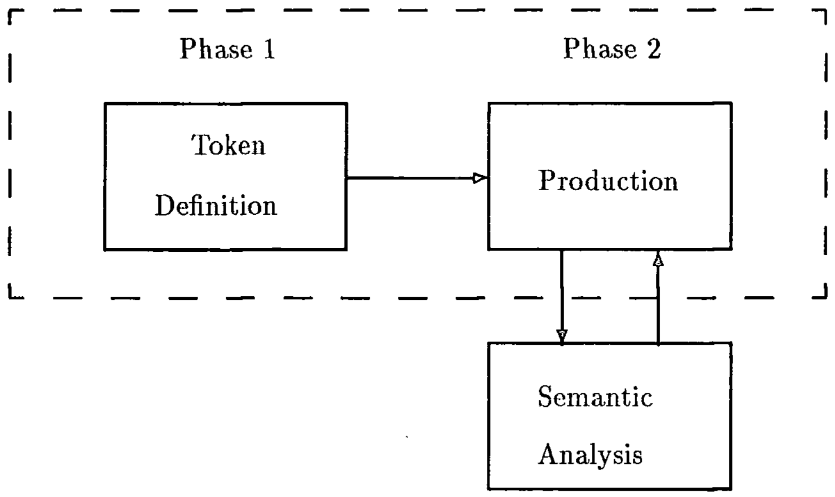

Figure 6.3: Parser Generation

In the second phase, the grammar of the Z specification given in [43] is defined. A production in YACC has the form:

non-terminal : right hand side { actions };

Each grammar rule given in [43] should be converted from BNF notation into the format expected by YACC as outlined below:

- The BNF define symbol "::=" becomes ":".

- The BNF concatenation symbol "," is simply omitted.

- An "*underscore*" character is substituted for each space that occurred within the *meta* identifiers.

- References to terminal symbols become references to the corresponding tokens provided by the scanner, for example:

  "begin{schema}" becomes BEGIN_SCHEMA;

  "end{schema}" becomes END_SCHEMA.

- An additional rule, of the following form, is written for all optional syntactic items:

  rule-name : *[* optional_items *]* ; will become

  rule-name : optional items

  | ;

- For all syntactic items that may occur one or more times, an additional rule of the following form is written:

  rule-name : syntactic-items rule-name

  | syntactic-name;

- Any grammar rules that use parenthesis for grouping are rewritten in their expanded form, for example

  rule ::= a(b|c); will become

  rule : ab | ac;

The action part in YACC defines semantics routines. When a rule is matched, the corresponding semantics routine is called.

6.2.1 Shift-Reduce and Reduce-Reduce Conflicts

The original grammar expressed in [43] contains 103 rules, these rules are translated into a YACC grammar consisting of 219 rules. When this version of the grammar is used to produce a parser, YACC reports 90 reduce-reduce conflicts and 119 shift-reduce conflicts. The main reason for these conflicts is that the grammar of Z contains a great deal of ambiguity and is not suitable for parsing by an LALR(1) parser.

In principle, YACC applies two straightforward rules to resolve these two types of conflict:

- With shift-reduce conflicts, YACC will use a shift in favour of a reduction.
- With reduce-reduce conflicts, YACC will use the production declared first in the grammar definition.

Of the two types of conflict, reduce-reduce conflicts are important and usually indicate a probable error in the grammar. They arise because there are two or more possible grammar rules that can be applied to the same input sequence. Sometimes, rewriting of the grammar is needed to avoid reduce-reduce conflicts.

Shift-reduce conflicts can often be accepted as a rule of thumb and they can often be resolved using precedence and association. So they are disregarded until the reduce-reduce conflicts have been addressed.

Reduce-reduce conflicts When the original version of parser is executed, Yacc report 109 reduce-reduce conflicts. For example, the rules *schema_name* and *ident* are as shown below

```
schema_name     : word ;
word            : VARIABLE ;
ident           : word decoration ;
opt_stroke_list :    "'
                | '?'
                | '!'
                | SUB
                | ;
```

When rule *opt-stroke-list* is empty, a parsing conflict arose because, after reading a ';' the parser does not know whether to reduce the rule *schema_name* or *ident*. For this parsing conflict, the grammar is rewritten using a rule and a new token as follow:

```
schema_name     : sword ;
word            : SVARIABLE
word            : VARIABLE ;
ident           : word decoration ;
opt_stroke_list :    ' ' '
                | '?'
                | !'
                | SUB
```

```
            | ;
```

This change removes all reduce-reduce conflicts.

Shift-reduce conflicts After all the reduce-reduce conflicts have been removed attention is then focused on the shift-reduce conflicts.

In the declaration section of parser, the relative precedence and association for all the relevant terminal symbols are declared. This then automatically assigns precedence levels to certain grammar rules as follows:

Each rule is given the precedence level associated with the last terminal symbol mentioned in its components. Consequently, rules which do not contain a terminal symbol or whose last terminal symbol does not have a declared precedence, are not assigned a precedence level.

Parsing conflicts are then resolved by comparing the precedence of the grammar rule being considered with that of the look-ahead token:

1. If the precedence of the look-ahead token is higher then the parser will shift.
2. If the precedence of the rule is higher then the parser will shift.
3. If they have equal precedence then the choice is based on the association of that precedence level.
4. If neither the rule nor the look-ahead has precedence then the default is to shift.

After the relative precedence and association for all the relevant terminal symbols have been declared, 68 shift-reduce conflicts are removed, and the parser still has 41 shift-reduce conflicts. Consider the fragment of the original grammar for expression given below:

```
expression      : expression in_gen expression;
```

```
in_gen        : REL
              | PFUN
              | FUN
              | PINJ
              | INJ
              | PSURJ
              | SURJ
              | BIJ
              | FFUN
              | FINJ ;
```

It can obviously be seen that the rule *expression* does not contain a terminal symbol, consequently, it does not have an associated precedence level. Hence, YACC is unable to use precedence and association information to resolve the parsing conflicts. In order to resolve these parsing conflicts, the rule *expression* is rewritten as follow:

```
expression    : expression REL expression
              | expression PFUN expression
              | expression FUN expression
              | expression PINJ expression
              | expression INJ expression
              | expression PSURJ expression
              | expression BIJ expression
              | expression SURJ expression
              | expression FFUN expression ;
```

These changes remove another eight shift-reduce conflicts.

This method of inserting some terminal symbols is also used to enforce the predecence and association to the following rule

expression_1 : expression_1 in_fun expression_1.

These changes again remove another eight shift-reduce conflicts. But there are still 25 shift-reduce conflicts which have not be been removed. Because a shift in favour of a reduction can be used to resolve shift-reduce conflicts, the following rules are introduced:

item	: ident ADEF branch_list
def_lhs	: ident in_gen ident
var_name	: ident

With these rules, ten more conflicts are resolved. Finally, the remaining 15 parsing conflicts are also resolved by introducing similar rules as those three rules just introduced above.

6.2.2 Error Handling and Recovery

The parser generator must be capable of handling and recovering all syntax errors. During the parser generation, there are syntax errors in the overall structure of the program. Common examples are omitting a semicolon between statements or forgetting a closing section bracket. For the syntax errors, there are four types of error recovery which are possible to use in combination.

- Panic mode recovery — The input tokens are discarded until a token that signifies a consistent position is reached. This method is simple and will not get stuck in a loop.

- Phrase level recovery — Local alternations are made to the input tokens to obtain a valid phase that would allow parsing to continue. Inserting deleting, changing and swapping tokens are all possible. The method works poorly when the error occurred some way back in the input.

- Error productions — This method uses a grammar with productions to pick up some errors and construct a parser using this grammar.
- Global error correction — This method seeks to transfer a source program to a program that can be parsed correctly. This is far too expensive to be used.

YACC provides error handling which makes use of error productions. In this thesis, panic mode and error productions are used in combination. The general form is:

non-terminal : error *synchronising set*

The synchronising set is a set of symbols. On encountering an error, YACC discards input tokens until it finds one in the synchronising set, or, if there are non-terminals in the synchronising set, one that can eventually be reduced to one in the synchronising set. It can then shift the token and eventually reduce the error production, allowing parsing to resume.

The major difficulty associated with error recovery is the question of where to restart parsing after a syntax error has been deleted. If parsing restarts at an inappropriate phase then the first syntax error is likely to lead a whole stream of connected syntax errors.

Perhaps the most straightforward strategy relies on the presence of a single token used to mark the end of each statement. In this situation when a syntax error has been detected, all tokens are simply ignored until the end of statement marker is found and then restart parsing. Unfortunately the Z specification language does not have end of statement markers, which makes error recovery more complicated.

In this parser, the error recovery strategy is highlighted below:

- For each grammar rule, a synchronising set is defined. Some of the synchronising sets contain terminals and nonterminals and some are empty.
- For terminals in the synchronising set, the parser discards input tokens until it finds one in the synchronising set; for non-terminals in the synchronising set , it can eventually be reduced to one in the synchronising set, the parser can then shift the token and eventually reduce the error production, allowing parsing to resume.
- For the empty synchronising set, the parser can immediately reduce by error production.

To illustrate this strategy consider the following example. If an error occurs in the rule basic_decl then all tokens are discarded until one of the following is found:

- any of the nonterminal "ident", "schema-ref", "op_name", indicating the end of the current section of basic_decl;
- the token "WHERE", indicating the end of the decl-part.

The grammar rules that incorporate error recovery are listed in table 2 along with a list of the terminals and nonterminals which will cause parsing to resume.

unboxed_para	END_ZED
item	']', schema_name, ident, predicate, END_ZED
axiomatic-box	END_AXDEF
generic_box	END_GENDEF
decl_part	WHERE, ident, schema-ref, op-name
axiom_part	END_SCHEMA, END_AXDEFF, END_GENOEF, '}', NL, ALSO, PRE, TRUE, FLUSE, LNOT, FORALL, EXISTS, EXIST_1, schema_ref, '(', ')'
def_lhs	DDEF
schema_exp	END_ZED, FORALL, EXISTS, EXTST_1, '[', ']', '(', ')', LNOT, PRE
schema_text	ident, ']'
predicate	END_SCHEMA, END_AXDEFF, END_GENOEF, '}', NL, ALSO, PRE, TRUE, FLUSE, LNOT, FORALL, EXISTS, EXIST_1, schema_ref, '(', ')'
expression	POWER, pre_gen, INTEGY, MINUS, LANG, RANG, LBAG, RBAG, '(', ')', THETA '{', ']', LAMDA, MU
set-ref	'}"

Table 2: The Grammar Rules for Error Recovery

6.3 Semantics Analysis and Expected Output Generator

It is customary to distinguish between the syntax and the semantics of a programming language. The syntax is concerned with the grammatical structure of programs. The semantics is concerned with the meaning of grammatically correct programs [45].

Semantic analysis is concerned with the identifiers and constants that appear in a source program. "Literal" constants are usually recognised by the scanner. Their attributes are deduced from their forms. Thus for identifiers we must:

1. on encountering the declaration of each identifier, create a new identifier structure containing its attributes;

2. whenever the identifier subsequently occurs, locate the appropriate identifier structure and inspect its structure;

3. in certain circumstances, locate the identifier as in 2 above and update some or all of its attributes.

In this thesis, semantics analysis is concerned with constructing a test case table and getting expected output related to the test-domain from a specification written in Z. The diagram of semantics analysis is showed in Figure 6.4.

Specifications of large systems in Z are often built up by specifying smaller sub-systems using schema calculus. A schema of a module in Z can be expressed as:

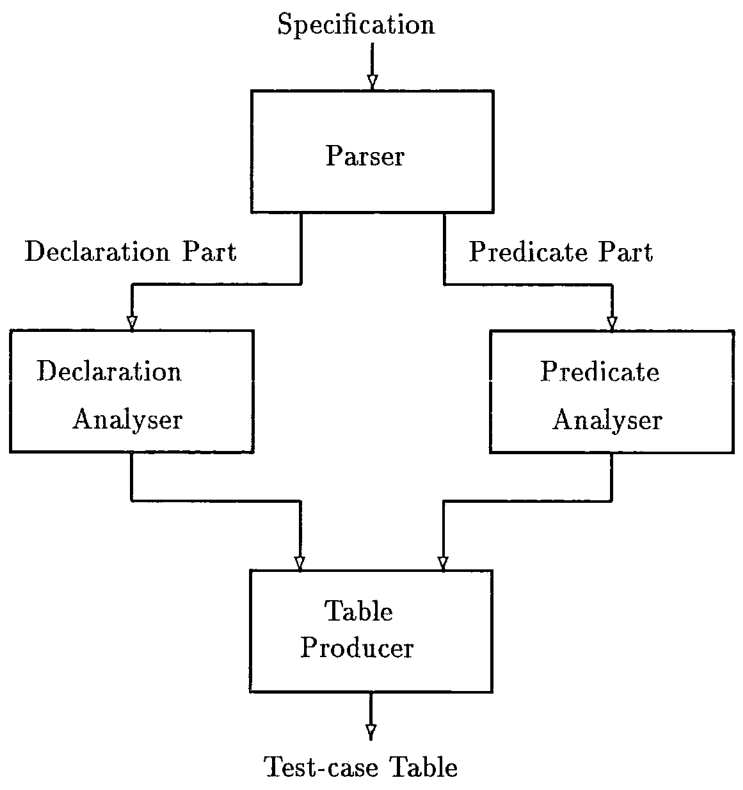

Figure 6.4: Semantic Analyser

$Schema - name$

$declaration - part$

$predicate - part$

In the declaration-part of the schema, all names of inputs, outputs and states are given. By convention names ending in ‘?’ are input data; names ending in ‘!’ are output data; names ending in ‘” are updated states; and names ending in ‘?’, ‘!’ and ‘” are otherwise just part of the name. A symbol table which holds names, values and types of input data, output and states are built as the following:

```
typedef struct  idnames {
        idptr   inext;
        typeptr itype;
        int     ivalue;
        char    iname[LENGTH+1];
        idptr   ileft;
        idptr   irght;
        } idnames
```

1. *inext* is used to temporarily chain together the identifiers in a decl-part.
2. *itype* points to the type structure for this identifier.
3. *ivalue* is its value.
4. *iname* is the spelling of this identifier.
5. *ileft* is the left link in the binary tree.
6. *iright* is the right link in the binary tree.

When the keyword WHERE is encountered, it indicates the end of the declaration-part and the beginning of the predicate-part. When a lookahead symbol is the keyword "WHERE", routine *column1()* is called to get the first column of the test case table. The predicate-part consists of a series of predicates. Test-domain and expect output are given from these predicates. The keyword LOR is a partition of the test domain. If there are m LOR in the predicate-part of the specification, then there are m+1 test-domain partitions. In each test-domain partition, a symbol table is built as the following:

```
typedef Struct tests {
        typeptr itype;
        int     ivalue;
        char    iname[LENGTH+1];
        idptr   ileft;
        idptr   irght;
        } tests.
```

1. *itype* points to the type structure for this identifier.
2. *ivalue* is its value.
3. *iname* is the spelling of this identifier.
4. *ileft* is the left link in the binary tree.
5. *iright* is the right link in the binary tree.

When a keyword LOR and END_SCHEMA is recognised, the routine test-domain() is called to get the test-domain and the expected output related to this test-domain. A test case table can be built using this information.

6.4 Summary

The implementation of the oracle prototype was described in this chapter, including the implementation of a scanner and a parser used for the Z specification, using Lex and YACC. This utilised an existing Fuzz tool to fulfill syntax checking and printing the Z specification in the box style. Parts of semantics analysis are also implemented for the Z specification. When a given form of the Z specification file is input to the prototype, the prototype can check the syntax of the Z specification and print out the Z specification in a standard box style, as well as printing out the expected outputs related to the input test domain. This is the form of oracle which was designed in this thesis. The corresponding test case can be obtained from the table.

However this prototype can only work on some simple textbook examples. Further research needs to be carried out for more complicated and practical examples. Nevertheless, it looks promising in that the Z Specification can be interpreted using a special semantics and the feasibility of this needs further exploration.

Chapter 7

Results and Evaluation

In this chapter, an example is first presented as the result of using the prototype, followed by an evaluation of the research is given. Finally the value of formal specifications in software testing is discussed.

7.1 An Example of the Use of the Oracle

In chapter 4, an example of a banking system for transferring money form source account to destination account was given. For that particular example, the input file of the specification can be written as:

```
\begin{schema}{Transfer}
  \Delta Bank \\
  amount?: \nat \\
  src?, det?: ACCT\\
  report!: MESSAGE
\where
  src? \neq dst?  \\
  bal(src?) \geq amount? \\
  bal' = bal \oplus \{src? \mapsto bal (src?) - amount?, \\
                              dst? \mapsto bal(dst?) + amount? \}  \\
  report!: "OK" \\
 \lor\\
  src? = dst?  \\
  report! = "Same \ account \ for \ src \ and \ dst" \\
 \lor\\
  src? \neq dst?  \\
  bal(src?) \leq amount? \\
  report! = "Not \ enough \ money \ in \ src" \\
\end{schema}
```

When this file is input to the system, if there is any syntax error in the file, the system will report the syntax error, otherwise, a specification and test case table are printed as follows respectively:

Transfer

$\Delta Bank$

$amount? : \mathbb{N}$

$src?, det? : ACCT$

$report! : MESSAGE$

$src? \neq dst?$

$bal(src?) \geq amount?$

$bal' = bal \oplus \{src? \mapsto bal(src?) - amount?,$

$dst? \mapsto bal(dst?) + amount?\}$

$report! : "OK"$

$\vee$

$src? = dst?$

$report! = "Same\ account\ for\ src\ and\ dst"$

$\vee$

$src? \neq dst?$

$bal(src?) \leq amount?$

$report! = "Not\ enough\ money\ in\ src"$

bal	**bal $\geq$ Amount**	**bal $\geq$ Amount**	**bal $<$ Amount**
Amount?	$\mathbb{N}$	$\mathbb{N}$	$\mathbb{N}$
src?	src? $\neq$ dst?	src? $=$ dst?	src? $\neq$ dst?
dst?	src? $\neq$ dst?	src? $=$ dst?	src? $\neq$ dst?
report!	OK	same account	not enough money
bal'	bal' $=$ bal $\oplus$ sth	bal' $=$ bal	bal' $=$bal

The bal' $=$ bal $\oplus$ sth in the above table is:

bal' = bal $\oplus$ {src? $\mapsto$ bal (src? - amount?, dst? $\mapsto$ bal(dst?) + amount? }

7.2 Using Test Case Table (Test Case Selection)

Having established the test-domains table as in the previous section, the next step will be to select typical test cases from the set. The process is:

1. Assign a unique number to each equivalence class,
2. Until all valid equivalence classes have been covered by test cases, cover as many of the uncovered value equivalence classes as possible,
3. Until all invalid equivalence classes have been covered by test cases, write a test case that cover one, and only one of the uncovered invalid equivalence classes.

For a Z specification, selection will be the chosen state. In the previous example on the banking system, the state *bal* of the system consists of the balance of each account is:

```
┌─ Bank ─────────────────────────────
│ bal : ACCT ⇸ N
└────────────────────────────────────
```

thus, we might select the state first to be

bal:

4256 $\rightarrow$ 200

8957 $\rightarrow$ 320 With this state, the first set of data can be chosen as below:

Testdata 1:

src? $= 4256 \rightarrow 200$

drc? $= 8957 \rightarrow 320$

Amount $:= 100$

Testdata 2:

src? $= 4256 \rightarrow 200$

drc? $= 4256 \rightarrow 320$

Amount $:= 100$

Testdata 3:

src? $= 4256 \rightarrow 200$

drc? $= 8957 \rightarrow 320$

Amount $:= 500$

A test case table is thus obtained:

	TD_1	**TD_2**	**TD_3**
bal	as above	as above	as above
amount	100	100	500
src?	2546	2546	2546
dst?	8957	2546	8957
report!	ok	same account	not enough money
bal'	$4256 \rightarrow 100$ $8957 \rightarrow 420$	not changed	not changed

From the above table, we know, that the expected outputs should be:

Testdata 1: report! =ok

bal':

src' $= 4256 \rightarrow 100$

dst' $= 8957 \rightarrow 420$

Amount := 100

Testdata 2:

report! = same account

bal' = bal not changed

Testdata 3:

report! = not enough money

bal' = bal not changed

Therefore, *report'* and *bal'* are expected outputs. Unfortunately, the expected output *bal'* must be worked out by hand for this system at present.

7.3 Evaluation

A prototype based on the research in this thesis has been implemented on a SUN workstation and experiments show that the prototype can work on examples like the one presented in the previous section. From the above example, we know, when the test data are selected from input condition, the expected output is automatically given in the oracle table.

For this prototype, it is a try-on to use Z formal specification to gain automatically the expected output – the oracle table. At present, it can only accept similar cases like the example just shown as input. It needs a further work to accept complex case as its input.

The advantage of this approach is the generation of an oracle which is functionally independent of any human decisions. This provides a strong foundation upon which a complete testing system can be built, i.e., by adding test case generation and gathering test coverage information. Additionally the system can provide motivation for generation of a formal specification during the software development cycle. The system integrates Z formal specification techniques with the process of software testing.

The implementation is independent of other tools, in particular a compiler. An oracle is generated to model the particular specification expressed, instead of requiring compiler extensions to drive test cases through the program. But the system can only automatically generate an oracle for small and comparatively simple Z specifications.

7.4 The Value of Formal Specification in Software Testing

A formal specification readily lends itself to test generation. It has the additional validation value of being implementation independent, as specifications are generally not written in a programming language. A functional testing approach has the advantage that testing oracles as well as test inputs are obtainable from the specification. A structural test approach would still require some form of specification for a test oracle.

The generation of a complete testing system from a formal specification would provide a greatly enhanced tool. A program specification provides rules for describing its inputs syntactically, which can provide a basis for the generation of test cases. An exhaustive approach which generates all possible inputs is not feasible. However, certain boundary information does appear to be present in the BNF grammars themselves. It may be possible to generate an interesting set of test cases and to build an automated testing system. The advantages of automated testing from a formal specification are as follows:

1. Test cases are available immediately after the specification is developed and

thus can be applied to incomplete or partially designed and coded programs. This in turn gives the ability to catch faults early in the development cycle and to reduce the amount of expensive recoding and redesign.

2. Automated testing from formal specifications does not require a human tester to have a complete understanding of the program specification or the code to generate test cases. This is important for a complex specification or for convoluted code.

3. Unlike most testing methods, automated testing from formal specifications does not require a tester to manually derive expected outputs for representative sets of test inputs. This is especially important for new systems for which no good test cases are known.

7.5 The Limitation of Formal Specification in Software Testing

There are some limitations when formal specification is used in software testing. The prototype built in this thesis cannot accept a complex example as input at the moment. A test oracle derived alternatively from Z specification means that Z formal notation have to be executed. It required that formal semantics will be defined for Z specification. Further work needs to be done in order to deal with more complicated examples.

To use Z formal notation as an oracle, the concrete input and output must be converted into their abstract representations. In many cases it would be difficult to get a test oracle without having other operations on the data type available. For more complex data types, the Z formal notation leads to be simpler.

This thesis has demonstrated how a test oracle for a simple example can be derived from its formal specification. It has been shown that specification-based testing must be further developed and should be incorporated into the software development lifecycle. this requires the use of formal specification languages in the specification and design phase and implementation. We intend to explore the possibility of exploring specification languages to incorporate test case descriptions that the user can specify the environment can generate automatically. We believe that developers will be less reluctant to use formal specification languages if we can demonstrate concrete advantages to be gained from their use in testing.

Chapter 8

Conclusions

8.1 Review of Project

Software testing has been identified as an expensive phase of the software life cycle. Therefore, research into the development of testing tools used by testers is needed. In particular, automated software testing tools for functional or structural testing are urgently needed.

A number of authors have suggested methods for functional testing, and there are also a substantial number of systems based on this approach. Functional testing depends on the availability of a test oracle used to determine the correctness of the output for a particular test input. A program can only be tested properly if the tester has the exact knowledge about what the program under test should and should not do. This justifies a need for a test oracle. Such information, for determining if a program is behaving correctly, can generally be derived from the specification of the program.

Practical issues of testing and the techniques of formal specification are often regarded as incompatible and irrelevant to one another. Some software testing techniques based on specifications make use of either informal specifications, or a specific specification which is written for software testing. This thesis has demonstrated that this need not be the case, by constructing a prototype oracle based on the Z formal specification language.

8.2 Assessment: Achievements

The following have been achieved during the project:

- An overall review of the software testing strategies and testing techniques was conducted, which analysed their strengths and weaknesses, and indicated the importance of functional testing in software testing.
- An overall survey of the testing oracles was carried out, which classified testing oracles, and assessed the existing oracles.
- A method was proposed for constructing an automatic oracle based on the Z formal specification language and a design was constructed.
- An investigation into relevant tools was done for constructing the proposed oracle, e.g. *fuzz*, Lex, YACC, etc.
- A parser was constructed for the Z specification language to develop an automatic testing oracle.
- a prototype of the testing oracle was constructed based on a formal specification in Z.
- experiments were carried out with the prototype.
- the results of the experiments were evaluated.

8.3 Assessment: Criticisms

An ideal system could translate a formal specification such as Z into an automatic correctness oracle. This automatic correctness oracle would validate the consistency of an unlimited number of program inputs and outputs with the specification. A complete tool implementation is beyond the resource constrains of this thesis. In this thesis, only a number of experiment examples have been considered. For the given syntax and a representation for the test-domain, the specification was parsed, then the attributed grammar style operations were used to develop the partition moving down and up the parse tree. The specification was interpreted to get some expected output related to test-domain using special semantics for Z. If we consider more different representations , it is not simple. There is a sense in which we must "execute" the specification. Superficially it seems promising, but needs exploration further to ascertain its feasibility. Even though the specification need not itself to be executed, it would still be possible to carry out the limited experiments from the execution required here.

8.4 Future Directions

It has been seen from the discussions in previous chapters that constructing an automatic oracle using formal specification is potentially of interest in software testing. To justify the arguments proposed in this thesis, further implementation of the tool and experimenting with using the tool will be the main direction for future research.

The formal semantics of Z notation is given in [41]. The formal semantics provides a foundation for a logical calculus for reasoning about a specification and deriving consequences from them. The successful application of formal methods in industry will be helped by a software tool [41]. Perhaps a future research project is to use the formal semantics of Z (such as denotational semantics and axiomatic semantics) to construct an automated testing tool.

The formal specification means that all the early parts of the testing procedure are easy to carry out. The functions have been identified, with their parameters (and the environment conditions, if they are regarded as different) so the first stages have already been done. The formal specification means that valid results for each test case can be worked out with certainty. Thus the formal functional testing will become important in software testing. Perhaps the most important future direction is the need to carry out future research into the construction of a completely automated tool based on a Z specification.

Chapter 9

References

[1] Abbott, J., *Software Testing Techniques*, NNC Publications, 1986.

[2] Aho, A. V. and Ullman, J. D., *Principles of Compiler Design*, Addison-Wesley Publishing Company, 1977.

[3] Avizienis, A. and Chen, L., "On the Implementation of N-version Programming for Software Fault-Tolerance during Program Execution ", Proceedings of COMPSAC Conference, 1977.

[4] Bell, G., Morrey, I. and Pugh, J., *Software Engineering — A Programming Approach*, Prentice-Hall INC., Englewood Cliffs, New Jersey, 1987.

[5] Bennett, J. P., *Introduction to Compiling Techniques: A First Course Using ANSI C, LEX and YACC*, The McGraw-Hill International, London, 1990.

[6] Bertolino, A., "An overview of Automated Software Testing", *J. System Software*, Vol. 15, pp. 133–138 (1991).

[7] Budd, T. A., Demillo, R. A., Lipton, R. J. and Sayward, J., "Theoretical and empirical studies on using program mutation to test the functional correctness of programs", Proc. ACM Symp. Principles of Prog. Lang., 1980.

[8] Budd, T. A. and Lipton, R. J., "Mutation Analysis of Decision Table Programs", Proc. Conf. Information Science and Systems, 1979.

[9] Camuffo, M., Maiocchi, M. and Morselli, M., "Automatic Software Test Generation", *Information and Software Technology*, Vol. 32, No. 5 (June 1990).

[10] Choquet, N., "Test Data Generation Using a Prolog with Constraints", Workshop on Software Test, Los Alamitos, CA, 1986.

[11] Clarke, L. A. and Richardson, D. J., "The application of error-sensitive testing strategies to debugging", *ACM SIGplan Notices* , Vol. 18, No. 8 (1983).

[12] Coward, P. D., "A review of software testing", *Information and Software Technology*, Vol. 30, No. 3 (1988).

[13] Davis, M. D. and Weyuker, E. J., "Pseudo-Oracles for Non-testable Programs", ACM, November 1981.

[14] Day, J. D. and Gannon, J. D., "A Test Oracle Based on Formal Specifications", 2nd Conf. on Software Development Tools, Techniques and Alternatives, December 1985.

[15] Duncan, A. G. and Hutchison, J. S., Using Attributed Grammars to Test Designs and Implementations, 1981.

[16] Fairley, R. E., *Software Engineering Concepts*, McGraw-Hill Book Company, 1985.

[17] Farmer, M., *Compiler Physiology for Beginners*, Charwell-Bratt Ltd, 1985.

[18] Frankl, P. G., "Tools for Testing Object-oriented programs", Proceedings of Pacific Northwest Software Quality Conference, 1990.

[19] Gerhart, S., "A Test Data Generation Method Using Prolog", Technical Report, 1985.

[20] Gilb, T., "A Comment on the Definition of Reliability", *ACM Software Engineering Notes*, Vol. 4, No. 3 (1979).

[21] Gilbert, S., *Software Design and Development Generation Method Using Prolog*, Science Research Associates, 1983.

[22] Girard, E. and Rault, J. C., "A Programming Technique for Software Reliability", IEEE Symposium on Software Reliability, 1973.

[23] Gorlick, M. M., Kesselman, S. F., Marotta, D. A. and Parker, D. S., "Mockingbird: A logical methodology for testing", *Journal of Logical Program*, Vol. 8, pp. 95–119 (1990).

[24] Gough, K. J., *Syntax Analysis and Software Tools*, Addison Wesley Publishing Company, 1988.

[25] Hall, P. A. V., "Relationship between Specifications and Testing", *Information and software technology*, Vol. 33, No. 1 (1991).

[26] Horebeek, I. V. and Lewi, J., *Algebraic Specifications in Software Engineering*, Springer-Verlag, Berlin , 1989.

[27] Horning, J. J., Lauer, H. C., Melliar-Smith, P. M. and Randell, B., "A Program Structure for Error Detection and Recovery", in *Lecture Notes in Computer Science*, Vol. 16, Springer, 1974, pp. 177–193.

[28] Howden, W. E., "Error, Testing Properties and Function Program Tests", in *Computer Porgram Testing*, North-Holland, 1981.

[29] Ince, D., "The Validation, Verification and Testing of Software", Technical Report, Computing Discipline Faculty of Mathematics, Open University, Milton Keynes, England, August, 1984.

[30] Jagota, A. and Rao, V., "TCL and TCI: A Powerful Language and Interpreter for Writing and Executing Black Box Tests", Proceeding of Pacific Northwest Software Quality Conference, Los Alamitos, CA, 1986.

[31] Mayrhauser, A. V., Software Engineering — Methods and Management, ACADEMIC PRESS, 1990.

[32] Meyers, G. J., *The Art of Software Testing*, John Wiley, 1979.

[33] Morell, L. J., "Unit Testing and Analysis", SEI Curriculum Module SEI-CM-9-1.1, Carnegie Mellon University, 1988.

[34] Panzl, D. J., "A Language for Specifying Software Tests", Proceedings of AFIPS National Computer Conference, 1978.

[35] Panzl, D. J., *Experience With Automatic Program Testing*, IEEE, 1981.

[36] Pesch, H., Schnupp, P., Schaller, H. and Spirk, A. P., "Test Case Generation Using Prolog", Proceedings of 8th International Conference on Software Engineering, Los Alamitos, CA, 1985.

[37] Randall, B., "System Structure for Software Fault Tolerance", *IEEE Transactions on Software Engineering*, Vol. 1, No. 4, pp. 220–232 (1975).

[38] Richardson, D. J., O'Mally, O. and Title, C., "Approaches to Specification-based Testing", *Software Engineering* , Vol. 14 , No. 8 (December, 1989).

[39] Schwartz, J. T., "Automatic Data Structure Choice in a Language of Very High Level", *CACM*, Vol. 18, pp. 722–728 (1975).

[40] Sommerville, I., *Software Engineering (3rd Edn.)*, Addison-Wesley Publishing Company, Wokingham, 1989.

[41] Spivey, J. M., *Understanding Z*, Cambridge University Press, 1988.

[42] Spivey, J. M., *The Z Notation*, Prentice Hall, 1989.

[43] Spivey, J. M., The fuzz Manual, 1992.

[44] Tsai, W. T., Volovik, D. and Keefe, T. F., "Automated Test Case Generation for Programs Specified by Relational Algebra Queries", *IEEE Transactions on Software Engineering*, Vol. 16 , No. 3 (March 1990).

[45] Watt, D. A., *Programming Language Syntax and Semantics*, Prentice-Hall International, Inc., 1991.

[46] Weyuker, E. J., "On Testing Non-testable Programs", *The Computer Journal*, Vol. 25, No. 4 (1982).

[47] Zelkowitz, M. V., Shaw, A. C. and Gannon, J. D., *Principles of Software Engineering and Design*, Prentice-Hall, Englewood Cliffs, NJ, 1979.

Appendix A

The Lexical Analyser of Z Specification

```
%{
/*scanner--lexical analysis */
#include        "y.tab.h"       /* generated by yacc */
#define         token(X)        X
extern  char    yytext[];
extern  int     yylval;
%}

delimiter       [ \t\n]
whitespace      {delimiter}+
alpha           [a-zA-Z_]
alphanum        [a-zA-Z_0-9]
ascii_char      [^\"\n]
escape_char     \\n|\\\"
digit           [0-9]
integer         {digit}+


%%


whitespace      {                 }
{alpha}{alphanum}        { return         lookup(VARIABLE); }
integer         { yylval = atoi(yytext) ;  return(INTEGER);}
"=="            { return        ADEF ;}
"::="           { return        DDEF ;}
"<>"            { return        NEQ ;}
```

```
"<="            { return        LEQ;}
">="            { return        GEQ;}
.               { return        yytext[0]; }

%%

/* reserved word */

#define         token(X)        X
extern  char    yytext[];

static  struct  keyword /* reserved word table */
        {
                char    *name;  /* representation */
                int     token_yylex;  /* yylex() value  */
        } keytable[]=    /* sorted  */
        {
                "\begin{zed}",          token(BEGIN_ZED),
                "\end{zed}",            token(END_ZED),
                "\begin{axdef}",        token(BEGIN_AXDEF),
                "\end{axdef}",          token(END_AXDEF),
                "\where",               token(WHERE),
                "\begin{schema}",       token(BEGIN_SCHEMA),
                "\end{schema}",         token(END_SCHEMA),
                "\also",                token(ALSO),
                "\bejin{gendef}",       token(BEGIN_GENDEF),
                "\end{gendef}",         token(END_GENDEF),
                "\defs",                token(DEFS),
                "\ldata",               token(LDATA),
                "\rdata",               token(RDATA),
                "\pre",                 token(PRE),
                "\forall",              token(FORALL),
                "\exists",              token(EXISTS),
                "\exists_1",            token(EXISTS_1),
                "\lnot",                token(LNOT),
                "\lpre",                token(LPRE),
                "\land",                token(LAND),
                "\lor",                 token(LOR),
                "\implies",             token(IMPLIES),
                "\iff",                 token(IFF),
                "\project",             token(PROJECT),
                "\hide",                token(HIDE),
                "\semi",                token(SEMI),
                "\true",                token(TRUE),
"\false",             token(FALSE),
"\in",                token(IN),
"\Xi",                token(XI),
"\Delta",             token(DELTA),
"\lambda",            token(LAMBDA),
"\mu",                token(MU),
"\theta",             token(THETA),
"\power",             token(POWER),
"\limg",              token(LIMG),
"\rimg",              token(RIMG),
"\langle",            token(LANGLE),
"\rangle",            token(RANGLE),
"\lbag",              token(LBAG),
"\rbag",              token(RBAG),
"\bsup",              token(BSUP),
"\esup",              token(ESUP),
"\notin",             token(NOTIN),
"\empty",             token(EMPTY),
"\subseteq",          token(SUBSETEQ),
"\subset",            token(SUBSET),
"\neq",               token(NEQ),
"\leq",               token(LEQ),
"\geq",               token(GEQ),
"\partition",         token(PARTITION),
"\inbag",             token(INBAG),
"\disjoint",          token(DISJOINT),
"\power_1",           token(POWER_1),
"\cross",             token(CROSS),
"\cup",               token(CUP),
"\cap",               token(CAP),
"\cat",               token(CAT),
"\setminus",          token(SETMINUS),
"\bigcup",            token(BIGCUP),
"\bigcap",            token(BIGCAP),
"\finset",            token(FINSET),
"\finset_1",          token(FINSET_1),
"\rel",               token(REL),
"\mapsto",            token(MAPSTO),
"\upto",              token(UPTO),
"\uplus",             token(UPLUS),
"\div",               token(DIV),
"\mod",               token(MOD),
"\filter",            token(FILTER),
"\dom",               token(DOM),
                "\rag",                     token(RAG),
                "\comp",                    token(COMP),
                "\circ",                    token(CIRC),
                "\dres",                    token(DRES),
                "\rres",                    token(RRES),
                "\ndres",                   token(NDRES),
                "\nrres",                   token(NRRES),
                "\oplus",                   token(OPLUS),
                "\plus",                    token(PLUS),
                "\star",                    token(STAR),
                "\beq",                     token(BEQ),
                "\pfun",                    token(PFUN),
                "\fun",                     token(FUN),
                "\pinj",                    token(PINJ),
                "\inj",                     token(INJ),
                "\psurj",                   token(PSURJ),
                "\bij",                     token(BIJ),
                "\surj",                    token(SURJ),
                "\ffun",                    token(FFUN),
                "\finj",                    token(FINJ),
                "\inv",                     token(INV),
                "\plus",                    token(PLUS),
                "\nat",                     token(NAT),
                "\nat_1",                   token(NAT_1),
                "\num",                     token(NUM),
                "\sub",                     token(SUB),
        };


lookup  (t)
int     t;
{
        register  struct keyword  *p;

        p = keytable;

        while (p->name)
                if (!strcmp(yytext), p->name)
                        return p->token_yylex;
                else
                        p++;

        return t;
}
```

Appendix B

The Semantic Analyser of Z Specification

```
/* parser for Z specification */


%{
#include        <stdio.h>
#include        <ctype.h>
#include        "lex.yy.c"

int             yylval;

%}

%token          BEGIN_ZED
%token          END_ZED
%token          BEGIN_AXDEF
%token          END_AXDEF
%token          WHERE
%token          BEGIN_SCHEMA
%token          END_SCHEMA
%token          ALSO
%token          VARIABLE
%token          BEGIN_GENDEF
%token          END_GENDEF
%token          FORALL
%token          DEFS
%token          BBAR
%token          LDATA
%token          RDATA
```

```
%token          PRE
%token          ADEF
%token          DDEF
%token          EXISTS
%token          EXISTS_1
%token          LNOT
%token          LPRE
%token          LAND
%token          LOR
%token          IMPLIES
%token          IFF
%token          PROJECT
%token          HIDE
%token          SEMI
%token          TRUE
%token          FALSE
%token          IN
%token          XI
%token          LAMBDA
%token          MU
%token          DELTA
%token          THETA
%token          POWER
%token          LIMG
%token          RIMG
%token          LANGLE
%token          RANGLE
%token          LBAG
%token          RBAG
%token          BSUP
%token          ESUP
%token          NOTIN
%token          SUBSETEQ
%token          SUBSET
%token          NEQ
%token          LEQ
%token          GEQ
%token          PARTITION
%token          INBAG
%token          DISJOINT
%token          POWER_1
%token          CROSS
%token          CUP
%token          CAP
```

```
%token          SETMINUS
%token          NL
%token          BIGCUP
%token          BIGCAP
%token          FINSET
%token          FINSET_1
%token          MAPSTO
%token          UPTO
%token          UPLUS
%token          DIV
%token          MOD
%token          FILTER
%token          DOM
%token          RAG
%token          COMP
%token          CIRC
%token          DRES
%token          RRES
%token          NDRES
%token          NRRES
%token          OPLUS
%token          PLUS
%token          STAR
%token          BEQ
%token          REL
%token          PFUN
%token          FUN
%token          PINJ
%token          INJ
%token          PSURJ
%token          BIJ
%token          SURJ
%token          FFUN
%token          FINJ
%token          INV
%token          PLUS
%token          CAT
%token          EMPTY
%token          NAT
%token          NAT_1
%token          NUM
%token          DCAT
%token          INTEGER
%token          SUB
```

```
%token          MINUS
%token          SVARIABLE

/* precedence */

%left           MAPSTO
%left           UPTO         ESUP
%left           '+'          '-'          CUP          SETMINUS
                CAT          UPLUS
%left           '*'          DIV          MOD          CAP
                COMP         CIRC         FILTER
%left           OPLUS        DRES         RRES         NDRES
                NRRES
%left           NEQ          NOTIN        SUBSETEQ     SUBSET
                '<'          '>'          LEQ          GEQ
                PARTITION    INBAG
%left           REL          PFUN         FUN          PINJ
                INJ          PSURJ        SURJ         BIJ
                FFUN         FINJ
%left           LAND         LOR          IFF          PROJECT
                HIDE         SEMI


%right          INV          PLUS         BSUP         STAR
%right          MINUS        DISJOINT     LNOT         PRE
%right          POWER_1      ID           FINSET       FINSET_1
                SEQ          SEQ_1        ISEQ         BAG
%right          IMPLIES


%start          specification

%%

/* rules section */


specification   : paragraph_list ;

paragraph_list  : paragraph
                | paragraph paragraph_list ;

paragraph       : unboxed_para
```

```
                  | axiomatic_box
                  | schema_box
                  | generic_box ;

unboxed_para      : BEGIN_ZED
                  item_sep_list item
                  END_ZED

item_sep_list : item
                  | item sep item_sep_list ;


item              : '[' ident_list ']'
                  | schema_name opt_gen_formals DEFS schema_exp
                  | def_lhs DDEF  expression
                  | ident ADEF branch_list
                  | predicate ;

ident_list        : ident
                  | ident ',' ident_list ;

opt_gen_formals : gen_formals
                  | ;

branch_list       : branch
                  | branch BBAR branch_list ;

axiomatic_box : BEGIN_AXDEF
                decl_part
                    opt_axiom_part
                    END_AXDEF;

schema_box        : BEGIN_SCHEMA '{'schema_name'}' opt_gen_formals
                    decl_part
                    opt_axiom_part
                    END_SCHEMA ;


generic_box : BEGIN_GENDEF opt_gen_formals
                    opt_axiom_part
                    END_GENDEF ;

opt_axiom_part : WHERE
                    axiom_part
                | ;

decl_part       : basic_decl_list ;

basic_decl_list : basic_decl
                | basic_decl sep basic_decl_list ;

axiom_part      : predicate_list ;


predicate_list : predicate
                | predicate sep predicate_list ;

sep             : ';'
                | NL
                | ALSO ;

def_lhs         : var_name opt_gen_formals
                | pre_gen ident
                | ident in_gen ident ;

branch          : ident
                | var_name LDATA expression RDATA ;


schema_exp      : FORALL schema_text '@' schema_exp
                | EXISTS  schema_text '@' schema_exp
                | EXISTS_1 schema_text '@' schema_exp
                | schema_exp_1 ;

schema_exp_1 : '[' schema_text ']'
                | schema_ref
                | LNOT schema_exp_1
                | PRE schema_exp_1
                | schema_exp_1 LAND schema_exp_1
                | schema_exp_1 LOR schema_exp_1
                | schema_exp_1 IMPLIES schema_exp_1
                | schema_exp_1 IFF schema_exp_1
                | schema_exp_1 PROJECT schema_exp_1
                | schema_exp_1 HIDE '(' decl_name_list ')'
                | schema_exp_1 SEMI schema_exp_1
                | '(' schema_exp ')' ;
```

```
schema_text     : declaration opt_predicate ;

declaration     : basic_decl_list


opt_predicate : '|' predicate
                | ;

schema_ref      : schema_name decoration opt_gen_actuals ;

opt_gen_actuals : gen_actuals
                | ;


basic_decl : decl_name_list ':' expression
                | schema_ref ;



predicate       : FORALL schema_text '@' predicate
                | EXISTS schema_text '@' predicate
                | EXISTS_1 schema_text '@' predicate
                | predicate_1 ;


predicate_1     : expression_rel_list
                | pre_rel expression
                | schema_ref
                | PRE schema_ref
                | TRUE
                | FALSE
                | LNOT predicate_1 %prec LNOT
                | predicate_1 LAND predicate_1
                | predicate_1 LOR predicate_1
                | predicate_1 IMPLIES predicate_1
                | predicate_1 IFF predicate_1
                | '(' predicate_1 ')' ;

expression_rel_list: expression rel expression
                | expression rel expression_rel_list ;


rel             : '='
                | IN
```

```
                 | in_rel ;

decl_name_list : decl_name
                 | decl_name ',' decl_name_list ;

expression_0     : LAMBDA schema_text '@' expression
                 | MU schema_text opt_s_expression
                 | expression ;

opt_s_expression: '@' expression
                 | ;

expression       : expression REL expression
                 | expression PFUN expression
                 | expression FUN expression
                 | expression PINJ expression
                 | expression INJ expression
                 | expression PSURJ expression
                 | expression BIJ expression
                 | expression SURJ expression
                 | expression FFUN expression;
                 | expression FINJ expression
                 | expression_1_list ;
                 | expression_1

expression_1_list: expression_1 CROSS expression_1
                 | expression_1 CROSS expression_1_list ;

expression_1     : expression_1 CUP expression_1
                 | expression_1 CAP expression_1
                 | expression_1 SETMINUS expression_1
                 | expression_1 MAPSTO expression_1
                 | expression_1 UPTO expression_1
                 | expression_1 UPLUS expression_1
                 | expression_1 DIV expression_1
                 | expression_1 MOD expression_1
                 | expression_1 FILTER expression_1
                 | expression_1 COMP expression_1
                 | expression_1 CIRC expression_1
                 | expression_1 DRES expression_1
                 | expression_1 RRES expression_1
                 | expression_1 NDRES expression_1
                 | expression_1 NRRES expression_1
                 | expression_1 OPLUS expression_1
```

```
                | expression_1 CAT expression_1
                | expression_1 '+' expression_1
                | expression_1 '-' expression_1
                | expression_1 '*' expression_1
                | POWER expression_3
                | pre_gen expression_3
                | MINUS expression_3 %prec MINUS
                | expression_3 LIMG expression_0 RIMG
                | expression_2 ;

expression_2    : expression_2 expression_3
                | expression_3 ;

expression_3    : var_name opt_gen_actuals
                | number
                | set_exp
                | LANGLE opt_expression_list RANGLE
                | LBAG opt_expression_list RBAG
                | '(' expression ',' expression_list ')'
                | THETA schema_name decoration
                | expression_3 '.' var_name
                | expression_3 post_fun
                | expression_3 BSUP expression ESUP
                | '(' expression_0 ')' ;


opt_expression_list: expression_list
                |  ;

expression_list : expression
                | expression ',' expression_list ;



set_exp         : '{' opt_expression_list '}'
                | '{' schema_text opt_s_expression '}' ;

ident           : word decoration ;


decl_name       : ident
                | op_name ;

var_name        : ident
```

```
                 | '(' op_name ')' ;

op_name          : '_'in_sym '_'
                 | pre_sym '_'
                 | '_' post_sym
                 | '_'LIMG'_'RIMG
                 | '-' ;

in_sym           : in_fun
                 | in_gen
                 | in_rel ;

pre_sym          : pre_gen
                 | pre_rel ;

post_sym         : post_fun ;


decoration       : opt_storke_list ;

gen_formals      : '[' ident_list ']' ;

gen_actuals      : '[' expression_list ']' ;

opt_storke_list : '''
                 | '?'
                 | '!'
                 | SUB
                 | ;

schema_name      : sword ;

sword            : SVARIABLE;

word             : VARIABLE ;

number           : INTEGER;

in_fun           : MAPSTO
                 | UPTO
                 | '+'
                 | '_'
                 | CUP
                 | SETMINUS
```

```
                       | CAT
                       | UPLUS
                       | '*'
                       | DIV
                       | MOD
                       | CAP
                       | COMP
                       | CIRC
                       | FILTER
                       | OPLUS
                       | DRES
                       | RRES
                       | NDRES
                       | NRRES ;

post_fun : INV
                       | PLUS
                       | ESUP
                       | STAR ;


in_gen                 : REL
                       | PFUN
                       | FUN
                       | PINJ
                       | INJ
                       | PSURJ
                       | SURJ
                       | BIJ
                       | FFUN
                       | FINJ ;

pre_rel                : DISJOINT ;


in_rel                 : NEQ
                       | NOTIN
                       | SUBSETEQ
                       | SUBSET
                       | '<'
                       | '>'
                       | LEQ
                       | GEQ
                       | PARTITION
```

```
                    | INBAG ;


pre_gen             : POWER_1
                    | ID
                    | FINSET
                    | FINSET_1
                    | SEQ
                    | SEQ_1
                    | ISEQ
                    | BAG ;

%%


main()
{
yyparse();

}
```