UNIVERSITY OF NOTTINGHAM
SCHOOL OF COMPUTER SCIENCE AND IT

The Automatic Assessment of Z Specifications

By
Zarina Shukur B.Sc(Hons)

Thesis submitted to the University of Nottingham for the degree of Doctor of Philosophy

January 1999

Ethos - Thesis for digitisation

Thesis details: Shukur, Zarina

'The Automatic Assessment of Z Specifcations'

Please exclude the following sections/pages:

Pages: 17, 18, 27, 133

To

my family

Acknowledgement

I would like to express my enormous appreciation to Dr Edmund Burke and Dr Eric Foxley for their advice, encouragement, and supervision particularly in writing the thesis.

I would also like to thank Professor Peter Ford, the department's head of School of Computer Science and IT, as well as Learning Technology Research Group for allowing me to attend several academic events both in the UK and abroad.

Thanks also go to the Malaysian Government for financial support during this project.

The statistical study is carried out using the information from many books and internet sources which it is not practical to list in this thesis. I also consulted with several friends; Brother Hassan, Liza and Rina (majoring in statistics) and would like to thank them for their help.

Last but not least, I would like to thank all my friends for their help directly or indirectly throughout my studies.

Abstract

The need to automate the process of assessing a specification in a learning environment is identified to be one of the fundamental ways to improve the use of formal notation in specifying a real system.

General issues involved in building an automatic marking system for computer-based courses are explored. Techniques that have been proposed for assessing a specification are also discussed. By considering the issues and the techniques, we describe how they can be used to build a system that is able to give a quality grade to a specification that is written in the Z language.

In the system, four quality factors are taken into consideration; maintainability of a specification (which considers the typographic arrangement of a specification and the specification complexity), and correctness of a specification (which reflects the static correctness and the dynamic correctness of a specification).

By using suitable quality metrics for specification maintainability, the results that are produced are compared to some values which can either be absolute values or relative to the model answer. The marks awarded for this factor are based on this comparison. Static correctness is carried out by applying a syntax and type checker. The marks granted for this factor depend on the outcome of the checker. Dynamic correctness is determined by employing a testing technique. In the context of a specification, the behaviour of a system-state, which is represented by so-called state variables, is analysed. The specification is 'executed' by using animation. The marks are given according to the correctness of the output and the final state.

The system is implemented within the well-known courseware management system, *Ceilidh*. There are fundamental differences between Z specifications, and the subject matter of other courses taught using the Ceilidh system (which are mostly computer programming courses). For this reason we take some time in this thesis to explain (in

some detail) the incorporation of the system within Ceilidh. The need for the fundamental components (i.e the editor, the syntax and type checker, the animator and the automatic marker) are discussed and described.

The system has been used by a group of 13 students who attended a Z course within the School of Computer Science and Information Technology at the University of Nottingham during the 1997/1998 academic year. The students were given a questionnaire about the system. An analysis of these questionnaires shows that the currently implemented tools are beneficial and helpful to the students. We also test the results of the system and compare them with a small selected group of human markers. The testing reveals very encouraging results and shows that the system can mark student scripts with a good degree of accuracy.

We conclude that this system can provide a very useful aid for teachers of the Z Specification language.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1. Motivation for automatically assessing a specification

Formal methods are taught in many institutions (in one form or another) on computing degree courses.[1] In software development, Wordsworth[2] describes them as "methods that exploit the power of discrete mathematics." By using formal methods, developers can make a precise record of something that can often be left vague and imprecise.[2] In the classic software life cycle model (sometimes called the 'waterfall model'), formal methods are primarily applied at the design stage as shown in Figure 1-1 which is adapted from[3] . In the model, it can be seen that the testing is done at a later stage, i.e after the coding process is finished. However, many new models like one shown in Figure 1-2 which is adapted from[4] , have become more flexible. According to the model, every step of the cycle is monitored in order to ensure the quality of software during the whole process. Therefore errors can be reduced at a later stage.

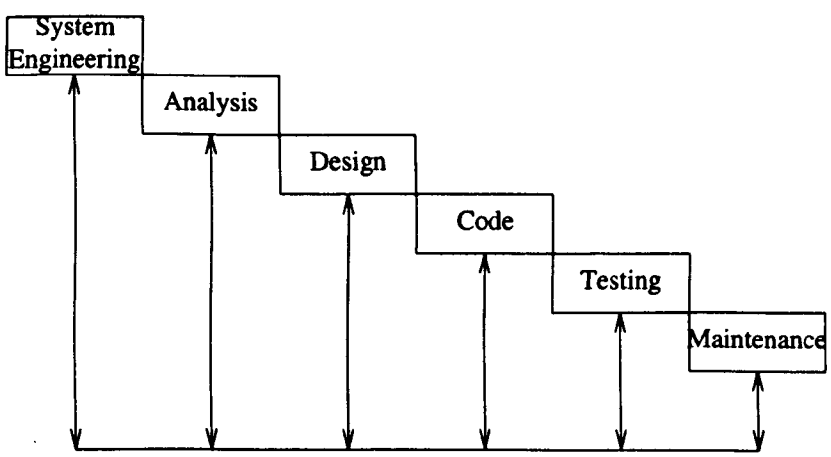


Figure 1-1 : Waterfall Model

This research focuses on controlling quality in the specification phase, by looking at fundamental problems in formal methods. The problems that are faced by formal

methods today are identified by Wordsworth[2] and are associated with educational issues. The teaching and learning of formal methods has been said to be a field of tension[5] for the following reasons.

- Formal methods are not sufficiently mature enough to be useful to the practising software engineer.[6]
- The lack of a coherent body of widely applicable, formal methods makes it difficult for educators to know what and how to teach existing techniques.[6]
- The mathematics being used in formal methods is a problem, since computer science lecturers are not good at teaching mathematics, and computer science students are not good at understanding it.[2]

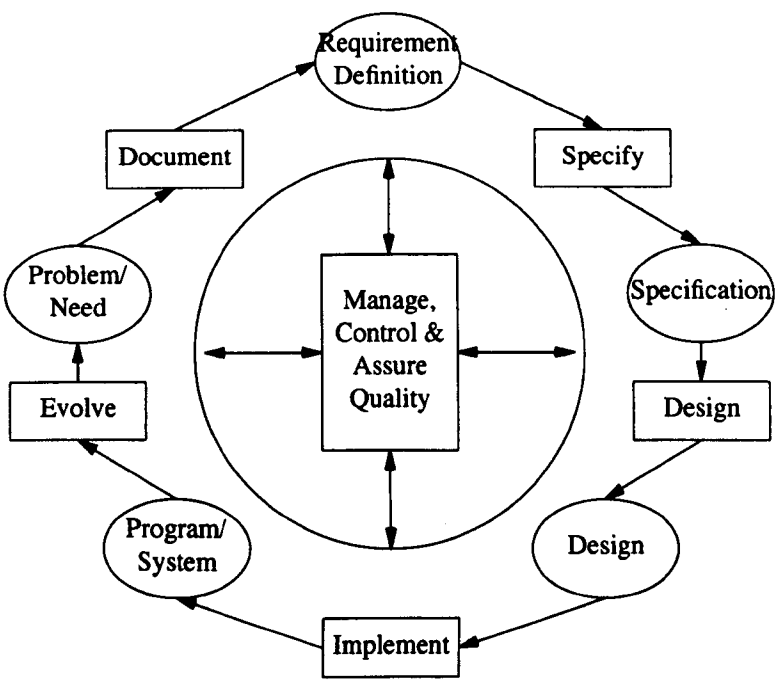


Figure 1-2 : Alternative Model of Software Life Cycle

There has been a recognition in recent years that education in the use of formal methods is vital, and that the role of mathematics is central to this.[7,8] As would be expected, many different approaches to teaching formal methods are applied in different higher education institutions.[6] They either:

- base an entire curriculum around a specific formal method, where one would learn one technique and apply the skill to software development acitivities;
- adopt a single course specifically devoted to formal methods, where a number of different formal techniques are introduced;
- integrate formal methods across the entire curriculum, learning several formal techniques in depth and apply the appropriate technique to the problem; or
- avoid teaching formal methods at all.

Even though it has been argued that the emphasis in teaching formal methods should be upon modelling[6,9] , it is important to express the model using some notation.[9] In general the skills that should be gained by the student include the ability:

- to model a problem using formal methods,
- to specify a model using formal methods techniques,
- to write a correct formal specification, and
- to read a formal specification.

Two main aspects to be considered when assessing student solutions to formal specification coursework are therefore:

- the quality of the modelling, and
- the quality of the writing a formal notation.

Usually, interaction between the student and teacher is needed in delivering a high quality of modelling.[9] Direct teacher involvement is essential here. Although the emphasis is on modelling, research done by Finney[10] on 62 undergraduate and postgraduate students shows that "in general the students found it difficult to understand any of the very simple Z specifications". One of her conclusions is that "the familiarity with notation and structure that comes naturally to them takes time, training and practice to acquire." This can be seen as a skill problem. Assessing a formal notation can be laborious as it involves extensive use of formal notation. The teacher's effort and time should be directed to such high value aspects of teaching as

modelling. Assessing mathematical notation manually can be a waste of energy. Where needed we should optimise the teacher's involvement and where possible we should minimise the teacher's work.

Some aspects such as ensuring the correctness of the use of notation in a specification are more easily accomplished by computer programs than by humans. This thesis demonstrates an attempt to automate the processs of assessing Z specification by using an animation concept as the fundamental process of observing its correctness. An overall aim of this research project is that the system should be used within the learning environment of higher education institutions.

The focus of the thesis is the correctness of the specification. However, other quality factors are also taken into consideration. This research is influenced by other works on automatic marking systems for programs and the feasibility of animating a specification. As discussed in the next chapter, formal specification has many similar attributes (as well as major differences) to computer programming. As far as the author is aware, this project is the first attempt to mark formal specification exercises automatically.

1.2. The Z specification language

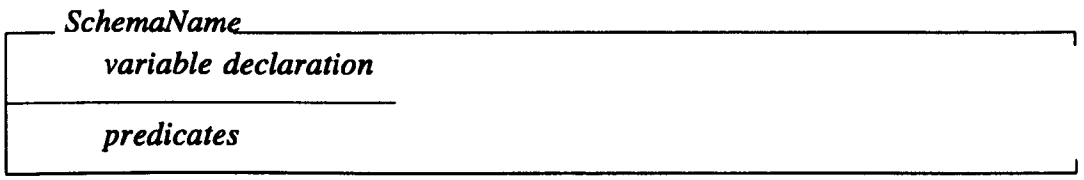
There are many types of formal specification that can be employed, including VDM, Z and the B method. The Z specification language is studied in this thesis. From a survey done by Austin and Parkin[1] on higher education institutions in United Kingdom, it was found that Z is the most frequently taught formal method. It is also one of the two most widely used formal methods in industry (the other is VDM). Furthermore, there are a number of very cheap or free support tools[9,11] such as type checkers and type-setters.

A great number of specification languages have been introduced over the years. Indeed, programming languages can be considered to be specification languages.

Basically, specification is divided into three types according to what is being specified; requirements specifications, functional specifications and design specifications.[12] Z is used to specify the functional phase, i.e a phase where we define the interfaces within the system. It is a model-based specification language that relies strongly on typed set theory and first-order predicate logic.

Z can be described in many ways.[13] It is called a language because it is used for communication. It is called a formal language because it has a precisely defined syntax. It is a formal specification language because the topic of communication is the specification and design of a software system. Its grammar is based on the mathematical fields of logic and set theory.

A Z specification normally contains state and operator schemas, as well as axiomatic descriptions. The schemas are often given names so that they may be referred to by other schemas. Each schema has the following form:



The schema name occurs at the top of the schema box. The contents of the box are separated by a short horizontal line, so that the *declaration part* of the schema lies in the upper region and the *predicate part* of the schema (containing one or more formulas) lies in the lower region.

The nature of Z specifications

Spivey[14] says, "in Z, schemas are used to describe both static and dynamic aspects of a system. The static aspects include:

- the states it can occupy

- the invariant relationships that are maintained as the system moves from state to state

The dynamic aspects include:

- the operations that are possible;
- the relationship between their inputs and outputs;
- the changes of state that happen."

Those aspects discussed by Spivey are described in mathematical notation. The data in a system is modelled by using *mathematical data types* and we describe the operations using *predicate logic.*

Schemas which describe static aspects of a system are known as *state schemas.* Burke and Foxley[15] said that state schema describes "the logic of the overall state of a system." The state is represented by some values held by some variables, i.e state variables.

Schemas which describe dynamic aspects of a system can be classified into two types; schemas that cause the system-state change (known as event or operation schema) and schemas that preserve the system-state but read information from it (known as observation or query schema). They both involve pre-conditions and post-conditions, specifying constraints before and after the event, and predicates relating to the state of the data after the event to the state before the event.

Throughout this thesis, we will consider a simple example called *telephone book* taken from the book *Understanding Z: A Specification language and its formal semantics* written by Spivey.[16] The example involves storing the names and telephone numbers of a group of people. The state schema is named *TelephoneBook*, the schema representing the addition of a new entry is *AddTelephone*, and the schema representing the retrieval of a named person's telephone number is *FindTelephone*. Those schemas might look like the following.

TelephoneBook

$known : \mathbb{P}\ NAME$
$telephone : NAME \pfun TELEPHONE$

$known = \mathbf{dom}\ telephone$

This state schema describes a telephone book system with two state variables i.e *known* and *telephone*. At any time, the names of people known to the database are exactly those for which a number is recorded.

AddTelephone

$\Delta TelephoneBook$
$name? : NAME$
$phone? : TELEPHONE$

$name? \notin known$
$known' = known \cup \{name?\}$
$telephone' = telephone \cup \{(name?, phone?)\}$

The $\Delta TelephoneBook$ indicates that we wish to use this schema in association with a state change. In any state change, a primed identifier indicates the value after the change, an unprimed identifier indicates the value before the change. The symbol ? in front of the identifier means that the identifier is an input to the operation. The above schema says that after the operation, a given name and telephone number will be in the telephone database with the condition that the name is previously unknown to the database.

FindTelephone

$\Xi TelephoneBook$
$name? : NAME$
$phone! : TELEPHONE$

$name? \in known$
$phone! = telephone\ name?$

The declaration $\Xi TelephoneBook$ means that we wish to use this schema in association with no change to the system data. The symbol ! means that the identifier

associated with it is an output of the observation. This schema describes the output of the query as the telephone number of a given name with the condition that the name must be in the database.

1.3. Types of exercise in a Z course

Different universities have different ways of teaching formal methods. In this section, we discuss the types of exercises which are given in the formal methods module offered by the School of Computer Science and Information Technology at University of Nottingham. The university teaches Z specification with the objective to expose the student to how to use Z in specifying a system.

The way the Z exercises are assessed is varied according to the focus of the question. As is the case with most courses, it will start with a simple problem, and the difficulty of exercises increases through the duration of the course. We divide the Z exercises into three categories: Exercises at the introductory level, intermediate level and advanced level.

- At the introductory level, the students are introduced to a Z specification in general, including various Z paragraphs and their syntax. The student will also usually carry out some revision on mathematical background. After completing this level, the student should be able to write simple Z paragraphs. The focus might be on writing a predicate for a specified problem.
- Students at the intermediate level are expected to have skills in Z syntax and layout. At this level, students will learn about several types of Z schema. The modelling technique for a simple problem would be introduced in this level.
- At the advanced level, the student will learn how the schemas are integrated by using schema algebra. The idea of getting program codes from the formal perspective might be presented. After finishing this level, the student is able to manipulate schemas and formally specify large software systems.

1.4. The aim of the research

The research described in this thesis, is aimed at developing a system to automate the assessment process of Z specifications. This system is designed (on purpose) to assess Z schema written by a beginner where much of the early coursework focuses on writing a specification, rather than on modelling. The aim of our system is not just to give a mark, but also to be able to reason about and interpret the marks produced. The experiment carried out on this system is intended to observe its performance and restriction. Furthermore, we recommend further studies on this problem. The system is named the Automatic Z Specification Assessment System (AZAS).

The attraction of doing this research is inspired by the feasibility of assessing programming language coursework automatically and the availabity of the basic tools (i.e type and syntax checker and animator) that share the same interface and support the basic function that are needed in this research. The experiment conducted on the system against a set of student answers has shown that it tends to behave like human.

1.5. Organisation of the thesis

This thesis consists of seven chapters (including this chapter).

- In chapter 2 we describe the influence of the evolution of automatic marking systems in computer based studies (which are mostly for computer programs). The fundamental principal of building an automatic marking system for a programming language and how this can be applied in building a system for assessing a specification language is discussed. We then show how we developed an automatic marking system for Z specifications.
- Chapter 3 concerns the detailed explanations of how the system is built by considering four aspects of quality: typographic, complexity, static correctness and dynamic correctness. For every factor, we explain how it is measured and the programs that are involved.

- The testing technique used in dynamic correctness is discussed in detailed in chapter 4. This involves the discussion of a debugging process.
- The description of the incorporation of the system into the existing courseware management system used in the University of Nottingham, (called Ceilidh[17]) is described in chapter 5. In addition, we also describe the implementation in the web pages.
- The performance of the automatic marking system is discovered by carrying out an experiment which involves sets of student answers as inputs to the system. The results produced by the system are compared to the results given by three human markers. This experiment is presented in Chapter 6. Further to this result, we present a discussion about the special requirements for the exercises to be assessed by the system.
- In chapter 7, we highlight the contribution that we have made and present some suggestions for further research.

References

1. S. Austin and G.I. Parkin, "Formal Methods: A Survey," Report, The National Physical Lab., Middlesex, 31 March 1993.
2. J.B. Wordsworth, "An industrial perspective on educational issues relating to formal methods," in *Teaching and Learning Formal Methods*, ed. C.N. Dean & M.G. Hinchey, pp. 1-9, San Diego, California, US, 1996.
3. R.S. Pressman, *Software Engineering*, McGraw-Hill Company Europe, 1992.
4. B. Ratcliff, *Introduction Specification Using Z : A Pracatical Case Study Approach*, McGraw Hill International, 1994.
5. E.W. Dijkstra, "Foreword," in *Teaching and Learning Formal Methods*, ed. C.N. Dean & M.G. Hinchey, San Diego, California, US, 1996.
6. D. Garlan, "Making formal methods education effective for professional software engineers," *Information and Software Technology*, vol. 37, no. 5-6, pp.

261-268, 1995.

7. J.B. Wordsworth, "Education in formal methods for software engineering," *Information and Software Technology*, vol. 29, Jan-Feb 1987.

8. "Educational Issues relating to Formal Methods," *Educational Issues Session of Z Users Meetings '94.*

9. C.N. Dean and M.G. Hinchey, "Introducing Formal Methods Through Role Play," *ACM SIGCSE Bulletin*, vol. 27, no. 1, pp. 302-306, March 1995.

10. K. Finney, "Mathematical Notation in Formal Specification: Too Difficult for the Masses?," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 158-159, February 1996.

11. C. Parker, "Z Tools Catalogue," ZIP/BAe/90/020, Software Technology Dept, British Aerospace, 10 May 1991.

12. D. Cooke, A. Gates, E. Demirors, O. Demirors, M.M. Tanik, and B. Kramer, "Languages for the Specification of Software," *Journal of System Software*, vol. 32, no. 3, pp. 269-308, 1996.

13. A. Diller, *Z An Introduction to Formal Methods,* John Wiley & Sons, 1994.

14. J.M. Spivey, *The Z notation : Reference Manual,* Prentice Hall, 1988.

15. E. Burke and E. Foxley, *Logic and its Applications,* Prentice Hall Europe, 1996.

16. J.M. Spivey, *Understanding Z: A Specification language and its formal semantics,* Cambridge University Press, 1988.

17. S. Benford, E. Burke, E. Foxley, N. Gutteridge, and A.M. Zin, "CEILIDIH: A Course Administration and Marking System," *Proceedings of International Conference in Computer based Learning in Science*, Vienna, 1993.

Chapter 2
Approaches to Automatic Assessment Systems

2.1. Introduction

Computer science courses should lead in using computer technology in teaching. Challenging areas, such as to automate marking processes, have attracted many people to do the research. These attempts focus on various types of computer program languages, from procedural, object-oriented, declarative to functional languages. Formal specification language, a functional specification, is an abstract specification language taught in computer science courses. In this chapter, we will consider several systems used for marking exercises in computer science courses. The main principle to develop such a system will be identified and following that, the approach that is taken to develop an automatic marking system for Z specifications will be described.

The idea in this chapter has been presented in the conference on Integrating Technology into Computer Science Education '97 and published in the respective proceedings.[1]

2.2. Automatic marking systems

Automatic program assessment systems have a long pedigree.[2] Foubister et al[2] touch upon the early automatic assessment system for computer programs in fair detail. They say that "many early applications were for the assessment of numerical analysis programs." They believe that "one of the earliest was developed in the late 1950's by Hollingsworth for IBM 650 machine code programs." In brief, these assessment processes take a student program (in the form of cards or paper tape) inline within a "grader", an assessment program, to provide initial variable values and make absolute checks on correct final values. Van Verth[3] wrote in her thesis that "until the mid-70s the emphasis in computer science courses was program correctness, concentrating on

the development of programs that ran correctly, i.e programs that produced correct answers, and that used resources such as time and memory economically."

However, Michaelson[4] diagnosed that "in the 1970's and early 1980's the focus moved away from correctness to optimality assessment." This movement is believed to be influenced by the awareness of the computer society concerning the need of discipline in the development of software, aiming at producing high quality software. This discipline, known as software engineering, was defined by Fritz Bauer[5] as:

> "The establishment and use of software engineering principles in order to obtain economically software that is reliable and works efficiently on real machines."

In this era, it can be seen that discussion about other factors of software has been greatly increasing. Discussion such as programming style by Kernighan in 1974, complexity by McCabe in 1976, the introduction of Halstead Software Science in 1977, and the broad classification of quality factors by McCall et al[7] in 1977 seem to have contributed to changing the ideas behind assessment systems. Furthermore, by having a software engineering discipline, we are not only considering the quality of a program, but consider quality in every step of the software life cycle.

Nowadays, automatic marking systems focus not just on the correctness of the program output, but analyse the output, the style of writing, the complexity and some other factors depending on the scheme of the system. Some systems are built to assess only one factor, whilst others assess a program by considering several selected factors. Van Verth[3] developed a system to assess the quality of a program in terms of complexity. Faidhi[8] built a program to analyse the complexity of Pascal programs. Some others include several quality factors in assessing programs. Zin and Foxley [9] built an assessment system, called Analyse, to mark C program by considering four factors, i.e layout, complexity, static correctness and dynamic correctness. Hung et al[10] developed ASSESS to mark factors in development effort, reliability, style, execution efficiency and complexity (which involves size, data structure and logic structure). A system for marking exercises written in a functional programming

language, Standard ML, which was introduced by Foubister et al,[2] assess functions for correctness and optimility (which involves style and type).

In brief, it can be said that the difference between assessment systems before and after the 80's is that more factors of assessment are taken into consideration. However, one aspect common to most of these systems is the use of teacher's knowledge.

2.3. Software quality factors

Pressman[11] said that "software quality is a complex mix of factors that will vary across different applications and the customers who request them." Schneidewind[12] defined it as a "degree to which software possesses a desired combination of attributes." The most common way of measuring software quality is by looking at many different factors which affect quality. These factors are then measured separately and finally combined to form the overall measure of the software quality. McCall et al[7] categorise factors that affect software quality into three main aspects:

- product operations: correctness, reliability, efficiency, integrity and usability
- product revision: maintainability, flexibility and testability; and
- product transition: portability, reusability and interoperability

The following is the description of the above factors in the context of a program.

Correctness. The extent to which a program satisfies its specification and fulfills the customer's mission.

Reliability. The extent to which a program can be expected to perform its intended function with required precision.

Efficiency. The amount of computing resources and code required by a program to perform its function.

Integrity. The extent to which access to software or data by unauthorized persons can be controlled.

Usability. The effort required to learn, operate, prepare input, and interpret output of a program.

Maintainability. The effort required to locate and fix errors in a program.

Flexibity. The effort required to modify an operational program.

Testability. The effort required to test a program to ensure that it performs its intended function.

Portability. The effort required to transfer the program from one hardware and/or software system environment to another.

Reusability. The extent to which a program (or parts of a program) can be reused in other applications.

Interoperability. The effort required to couple one system to another.

The factors proposed by McCall are very comprehensive. These factors are environment dependant and differ in their importance in the individual stages at the software life cycle. Some might not be appropriate to individual stages in the software life cycle.[13] However, the relative importance of different factors may vary in different environments.[13] For a small software project, Burgess[14] proposed that only four factors should be considered. These factors are; correctness, maintainability, usability and efficiency. In a study done by Zin[15] for programming quality in a learning environment, only correctness and maintainability are taken into consideration.

Software quality metrics

Whitty[16] said that "deriving and applying measurements for software systems is a thriving technical area of research, broadly termed 'software metrics'." A quality metric is a number that represents one facet of the quality factors. Pressman[11] recognised that "it is difficult, and some cases impossible, to develop direct measures of the above quality factors." He identifies two types of software quality factors; first, factors that can be directly measured (for example errors/unit time), second, factors

that can be measured only indirectly (for example usability and maintainability). In each case measurement must occur, and the software (such as documents or programs) must be compared to some data and an indication of quality will be arrived at. Factors which cannot be measured directly need a software metric to reflect them.

A lot of research about software metrics has been done. Most of it proposes ways to derive a metric to represent a value for the complexity factor, for example Halstead's Software Science[6] , Henry's Information Flow Metrics[17] and McCabe's Cyclomatic Complexity.[18]

There is a question of which software metrics should be used. Software metrics are said by Whitty[16] to be "an area particularly fraught with hazards and pitfalls, arising from poorly understood concepts in software engineering and doubtful analogies with other engineering disciplines." Kearney et al advised users of complexity measures to be "aware of the limitations of these measures and approach their applications cautiously".

Quality Indicators

As stated by Pressman above, these values (i.e quality metrics) will not give any result by themselves; a general indication of quality is needed. This can be done by conducting an analysis on a set of actual data and deduce which metrics are indicators. For example, McCabe furthered his study in complexity by analysing data from actual programming projects and concluded with the best cyclometric complexity metric. Another way is by using a model solution, assuming that the model is the best solution. This has been proposed by Redish and Smith.[19] Rees[20] mentioned that an assessor is required to provide a certain range of parameters as indicators. Hung et al[10] claimed to improve the scheme proposed by Rees. They overcame the subjectivity of choosing the parameters in Rees' approach (see below) by using the average measurement of a set of programs.

A quality factor such as correctness is determined correct if it can be proved to meet its specification. However, Michaelson[4] said that "most correctness assessment is empirical and based on checking a program's outputs from inputs".

2.4. Measuring quality

Measuring maintainability

Rees,[20] in his proposed measurement of programming style, compares the derived value with a set of parameters.

Figure 2-1 : Rees' approach for measuring programming style

The score for each factor is collected from the program. Based on each separate score a mark is given. The calculation for the mark is based on the scheme as shown in Figure 2-1, where the points have the following significance:

L: the point below which no mark is obtained.

S: the starting point of 'ideal' range.

F: the end point of 'ideal' range.

H: the point above which no mark is obtained.

Thus, scores between S and F obtain a maximum mark, those between L and S and between F and H are calculated by interpolation according to their exact position within the range, and those outside the range (L,H) receive no marks. The values for

L,S,F,H and the maximum mark for each factor are given as part of the assessment program.

Another approach was made by Redish and Smith,[19] where the marking of the programming style is based on a model program.

Figure 2-2 : AUTOMARK's approach for measuring programming style

The score for each complexity factor is calculated from the model program and student's program.

F: the non-negative values of the factors computed for the model program

T: the non-negative tolerences for the factor values F

M: the maximum mark available for each of the factors

W: the non-negative weight assigned to the mark for each factor

L,G: indicators taking one of the three values -1, 0, +1. This indicates whether being Less (L) or Greater (G) than F is worse (-1), about the same (0) or better (+1), respectively.

The calculation of marks for each factor is shown in Figure 2-2. The parameter F is taken from a model program, whilst M, T, W, L and G are decided by the teacher. If the score for one particular factor derived from the student's solution is within the interval F-T*F and F+T*F, then the student will be awarded 1 mark for that factor. If

the score is less than F-T*F, the mark will be calculated by linear interpolation (might be positive, negative or 0 depending on the value of L) between 0 and F-T*F. The same technique of calculation is also applied if the score is greater than F+T*F.

Measuring correctness

In general correctness tends to be classified either right or wrong. Awarding marks by strictly considering only these two extremes seems inefficient. It is preferable when looking at the correcness of student work if the mark awarded is based on the degree of correctness.

Zin and Foxley[9] proposed a way to measure program correctness by using an "oracle". The oracle involves a number of regular expressions to define the structures which it expects to find in the student program's output. They give an example, in a simple example of a program to convert centimeters to feet and inches, if the correct output value is 3 feet 4.69 inches, there might be two regular expressions. One would search for

"3" followed by "feet" or "ft"

and the other for

"4.69" or "4.7" or "5" followed by "inches" or "ins"

The score from a regular expression based oracle may be any value from 0 to 100%

Principally, four aspects should be considered when developing an automatic marking system. Firstly, to achieve high software quality in a system, the quality factors involved must be clearly defined, otherwise assessment of quality is left to intuition.[12] Secondly, we must define a way to derive the quality metrics. Schneidewind[12] signifies this issue by saying that "using metrics reduces subjectivity in software quality assessment by providing a quantitative basis for making decisions." Thirdly, the derived metrics have no meaning by themselves. They need

to be compared to other values to indicate quality. This is usually done by embedding teacher knowledge or research done by others. And lastly, a technique to allocate a mark should be considered.

Apart from that, an automatic grading system needs to meet certain other criteria[3] :

- A system must be able to evaluate those features of programs that are currently poorly defined;
- It should provide evaluations with which many graders would be in general agreement, distinguishing between programs demonstrating positive attributes and those demonstrating negative ones;
- It should be able to provide students with diagnostic information about poor programming practices and suggestions for improvement.

2.5. Approach for specification assessment

Software components are built using a programming language. We can say that programs are the end product of the software life cycle, even though the cycle itself might be a non-stop process. To keep the software up to the standard quality, the awareness of software quality in every stage of the life cycle should be stressed. Therefore, software quality factors are applied to all the software documents such as the program, specification, manual and other related documents. It has been proved that assessing a program automatically can be done by using those factors. We did this research to observe the feasibility of developing a system with the same concept as the previous automated marking systems. The subject to be assessed is a specification.

The perspective from which we approach the problem is that we wish to automatically assess the quality of Z specifications provided by students as part of their coursework for a Formal Specification course. The environment chosen for the assessment is the Ceilidh System, the most widely used student programming automatic assessment system in the world. This provides an environment into which

new assessment tools can be inserted, and which provides all the required administrative framework.

A program is said to be a special case of a specification.[21] It is written in more constrained style compared to the specification.

- Programs will react somehow to all possible inputs, whereas specifications need only cover inputs relevant to the intended context of use.
- Programs must be defined, whereas specifications may leave some cases ambigious or undetermined.
- Programs explicitly constrain outputs in terms of inputs, while specifications can constrain by any explicit or implicit means.[21]

Another way to describe this relation is as said by Hall,[22] "a specification and a program are defined as mappings f and P from some domain set D to some range set R."

specification: f:D -> R
program: P:D -> R

Therefore, we have chosen to apply similar techniques to those used in developing automatic marking system for programs in this study.

2.6. Quality in a specification

Several researchers have discussed definitions of quality of specification from different perspectives. Following Zin's[9] study for programming quality in learning environments, we have taken two quality factors (correctness and maintainability in a more general sense - see below) into consideration for use in assessing a specification.

2.6.1. Maintainability

Following Oman and Cook[23] , we divide the specification style into two categories: those pertaining to the typographic arrangement, and those measuring the structural content of the text, its complexity.

Typographics

Typographic style describes the way a specification text is presented. The importance of having style in writing has been touched by many researchers, such as stated by Rosalind[24] :

> "A consistent style helps people to read others' work, and can ease the reuse of parts of a specification."

There are two approaches to style: relative and absolute. Relative style is concerned with assessing something against an appropriate exemplar whereas absolute style is based on generic concepts. In the context of programs, the languages can be defined in terms of: a lexical analysis which specifies the symbol; syntax which specifies valid grammatical structure; semantics which specifies the meaning of well-formed symbol sequences; and pragmatics concerned with context of use. Michaelson[4] discussed programming style according to these four terms:

- Issues regarding the lexicon such as the length of identifier and the meaningfulness of identifier.
- Syntax is a subject of how well formed the sequence of symbols is, for instance issues of indentation and layout.
- Semantics dealing with the appropriate use of constructs, such as the fact that FOR constructs are better than WHILEs in certain cases.
- Pragmatics dealing with the context of use, for example, the use of comments, assertions and type annotations.

Various aspects of typographic style which are relevant to Z include: [24]

- naming conventions
- layout of the mathematics
- accompanying text
- Z style conventions
- document structure
- cross referencing and index conventions
- layout of proofs
- statement of proof obligations

Furthermore, the layout for a Z specification could be[25]

- define given sets, data types and constants
- define state variables
- define initial state
- define 'correct' operations
- define exceptional operations
- combine operation schemas
- compose component specifications to form a complete system specification

Complexity

Basili[26] defines complexity as a measure of the resources expended by a system while interacting with a piece of software to perform a given task. Software complexity has been researched by many people.[6, 17, 26, 27] Kearney et al[27] claimed that "the most widely known measures are those devised by Halstead and his colleagues that are collectively known as software science". Halstead's theory of software science is said by Curtis[28] to be "probably the best known and most thoroughly studied."

The Halstead measures[6] are functions of the number of operators and operands in the program. The basic elements of software science (which are called primitive measures by Pressman[11]) are

- the number of unique operators
- the number of unique operands
- the total number of operators
- the total number of operands

From these primitive measures, he derived functions for program length and program volume.

Additionally Kearney et al claimed that McCabe's theory regarding cyclomatic number has also received a great deal of attention. As has been concluded in their article in which they referred to research done by Basili[29] and Henry and Kafura,[17] "the cyclomatic number is strongly related to the Halstead metrics. Both are computed by counting lexical entities and it has been shown that the measures are highly correlated".

2.6.2. Specification correctness

If a formal specification is to be trusted, it should satisfy the following two conditions.

> Firstly, it must be well-formed. This means that statements in the specification must conform to the syntax and semantics of the specification language.
>
> The second condition, which is more important, is that the properties of the formal specification reflect the user requirements.[30]

Several techniques have been developed for measuring a specification.[11,30] The normal approach to validating a formal specification is by employing *formal reasoning*. However, Fields[31] has shown that this technique is very tedious and time consuming.

Another technique which can overcome some of these problems is *formal technical reviews* (FTR). This is a class of review techniques that includes walk through, inspection, round-robin and other small group technical assessments.[11] Each FTR is conducted as a meeting which focuses on a specific part of the overall specification. Before the meeting, copies of the specification are distributed to each reviewer. Each reviewer is expected to inspect the document, make notes, and become familiar with the work. During the meeting, the person who has developed the specification *walks through* it while reviewers raise issues based on their advance preparation. FTR is a very effective way for validating the specifications of technical documents. However, this approach is very informal and is labour intensive.[32]

Another technique which is more formal is the *viewpoint resolution* technique. This is a process which:

(i) identifies discrepancies between two different viewpoints,

(ii) classifies and evaluates those discrepancies, and

(iii) integrates the alternative solutions into a single representation.[33,34]

An advantage of the viewpoint resolution technique is that it has a level of formalising with all the inherent benefits. However, this technique requires several specifications to be produced by different people or groups of people.[30]

The *testing technique* for assuring the correctness of a formal specification executes the specification against test data. This idea has been proposed by many people, for example by Kemmerer,[35] Jalote[36] and Hall.[22] The different objectives of the testing mean that we have to create different methods for conducting it, especially in choosing test data sets. For the testing to be done, the formal specification must be executable. However, most formal specification notations are non-procedural and thus cannot be executed directly. This implies that before this type of formal specification can be tested, the specification must be translated into a procedural form which can be executed.

2.7. Other issue : Z tools

In order to make this study feasible, three basics tools have been identified needed to carry out the functions below.

- To automate the process of marking a Z specification it has to be in electronic form. As has been described before, Z specification consists of texts, mathematical symbols and lines. Therefore a way to represent it in electronic form is needed.
- A well-defined grammar exists for the Z language, therefore it is possible to check that every construct is well formed.[37] Similarly, every variable must be identified with a specific type. As with those programming languages which use types, it is possible to check that the usage of every variable conforms to its declared type. With Z, it is possible to automate both checks.
- To animate a formal specification document means to produce an executable prototype by (automatic) transformations which preserve correctness and other "interesting" properties.[35]

Many Z tools have been developed to support the above functions plus others which can be of great support in our study. However there are some problems that need to be considered.[38]

- Each tool typically involves a large initial investment of time to get to the point where the tool is useful.
- Few tools work together or share common interface designs, consequently, each tool must be learned from scratch and operated in isolation.

In our study, we use in-house tools which have been developed by researchers at the University of Nottingham. The important aspect of these tools is that they have the same interface design and they have the primary functions which are relevant to our study. Table 2-1 shows the functionality comparisons between the tools that we use (Tool 19) and other existing Z tools (Tool 1 - Tool 18). The information about the

tools is taken from the surveys made by Parker,[39] and by Steggles and Hulance.[40]

Table 2-1 : Functionality of Z Support Tools

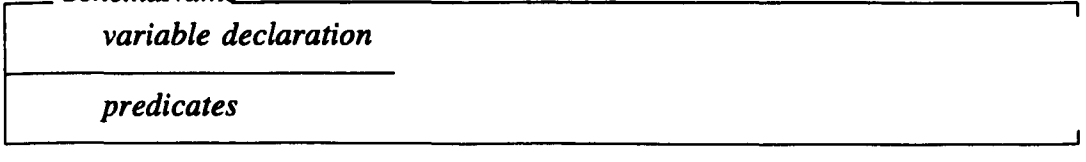
2.8. Design consideration

Before the possible assessment of specifications is discussed, some general points about assessment need to be made. These arise from experience in the use of Ceilidh for the assessment of similar topics.[41] Any assessment scheme which we eventually produce must satisfy a number of criteria.

- Its output must be helpful to the student. The output must not be, for example, simply a statement that their specification failed, or that they lost 10 marks. It should pinpoint the reasons for any losses in a form which acts to enhance the student's learning experience.
- It must be reasonably easy for the designer of the assessment system to set it up. In the exercise-specific information, the person setting up a new exercise should not need to know the workings of, for example, the Z to Prolog compiler. The marking procedures should be written in a high-level user-friendly language.
- The awarded mark scheme must allow flexibility, so that the teacher can concentrate marks in any area of the process, to suit the area currently being taught. The emphasis of the marking may change from week to week, and from course to course. In order to achieve this, the overall assessment must combine the sub-marks for each part of the assessment process by a formula which the teacher can easily change. In the present version of Ceilidh the teacher specifies weighting factors in a pre-determined marking formula combining the sub-marks; in the next version, much more general formulae will be permitted.
- The marking scheme must be incremental and progressive. By this we mean that the system must not merely award 100% for perfect answers and 0% for erroneous answers. It must be capable of awarding partial marks at the control of the teacher for partially correct solutions. In this case, the student should ideally be informed only of the most serious error, the error causing the most serious loss of marks in the current marking scheme. It would not help the educational process to give the student many messages concerning very minor losses of marks.

- A further point is that all specifications are written to represent the customer requirements, and the way the specification is written is dependent on the type of problem to be addressed. The evaluation of the specification quality must also take into consideration the requirements descriptions. In our system we will represent the "customer statement of requirements" by a specification model supplied by the teacher; the teacher presumably thinks that this is the best solution which reflects the requirements descriptions. This means that some aspects of the evaluation will be based on this supplied model.
- Apart from the (student) specification to be analysed, the system must therefore accept two more inputs, the set of values and formula with the control variables defining the importance of different factors in the assessment, and the model specification produced by the teacher (which is presumed by the teacher to represent accurately our "customer" statement of requirements). For the purpose of animation, the system must be supplied with a number of sets of states and queries. The number of queries deemed appropriate for testing the exercise will be determined by the teacher. This represents the input to the animation process. Also the system should be supplied with definitions of the expected answers that represent the output of the animation process; in Ceilidh these would be supplied as oracles.[42]

In the University of Nottingham we have a Z type checker that has been developed by Zin[30] which is called `zc`. Our assessment system uses `zc` to check the semantic and syntax of the specification. This requires that the specification be written in an ASCII file in a format developed at Nottingham, referred to as Z-roff[43] and originally designed as a pre-processor language for the Unix roff system. The Z specification



would be presented in roff format as:

.ZS SchemaName

variable declaration

.-

predicates

.ZE

The system developed in this study awards marks for specification maintainability using a similar technique to that technique proposed by Rees.[20] For the *typographic* category, we derive the metrics by analysing the specification, which is written in Z-roff format. Each selected typographic metric is marked according to a range of absolute values specified by the teacher to best reflect the good layout of the specification. The measurement of structural style (or complexity) is carried out based on the model solution, similar to the one proposed by Redish and Smyth[44] , since the structure of a specification depends on the type of problem being solved. Before we can compare the structural style of two specifications, we have to make sure that both specifications are equivalent, that is, they are solving the same problem. For the *complexity* category, the metrics are gathered by analysing a conceptual model file of the specification (which is created as part of a succesful application of *zc*). Each complexity metric is marked relative to the corresponding counts for the model specification.

The system measures the correctness of a Z specification by using two steps. First the specification semantics and syntax will be checked. The student is awarded full marks for this step if no error occurs. If errors occur, no further testing takes place. The overall mark for this step is categorised as *static correctness*. The next step is to ensure that the properties of the specification reflect the user requirements. A number of sets of test data are run against the specification. Each set has a weighting to reflect its importance. Marks are given according to the success of the testing against the test data. For a given test data, each aspect of success has its own weighting factor to reflect its importance.

To make the specification executable, we use the concept of *animation.* During the animation, the specification is 'executed' against a number of sets of test data. Not many tools have been developed to do animation; one such tool is the Zen and Zee (see Table 2-1). In the University of Nottingham, we have a tool named `zp`[30] which can form a basis for the animation of a Z specification. `zp` translates into Prolog a specification which has been successfully compiled by `zc`.

Although the concept of testing a specification has been used by Jalote[36] and Hall[22], the goal of their testing is different from ours. Jalote's[36] testing goal is to detect the incompleteness of the set of axioms, whilst Hall's[22] goal is to test implementations. The different objectives will create different methods of testing, especially in choosing test data sets.

Our testing objective is to test the degree to which the specification fulfills customer requirements. Our tests are therefore designed to see whether the specification will react in the way our customer wants if we give certain inputs to it. Since our system is meant for an educational environment, the teacher will act as the customer. The overall mark awarded for this step is categorised as *dynamic correctness.*

2.9. Conclusion

In this chapter we have presented the issues involved in building an automatic marking system for computer-based courses. Several techniques that have been proposed to be used in assessing specification were discussed. By considering these factors, we explained the idea of how we can develop a marking system for Z specifications. The idea shows that the development of an automatic assessment system for formal specifications written in Z is feasible and would provide a useful tool. Such a system would be of great help to anyone who has to assess the quality of specifications, and in particular for teachers of such courses. Ceilidh has been successful in the automatic on-line marking of programs but it needs to expand its capability to include other aspects of automatic marking. Supported by the existing Z

syntax checker and Z to Prolog translator, develop by Zin[30] , we have developed a system that can mark the quality of a software specification which is written in the Z language.

References

1. E. Foxley, O. Salman, and Z. Shukur, "The Automatic Assessment of Z Specification," *Proceedings of ITiCSE '97 Conference*, Uppsala, Sweden, June 1997.
2. S.P. Foubister, G.J. Michaelson, and N. Tomes, "Automatic assessment of elementary Standard ML programs using Ceilidh," *Journal of Computer Assisted Learning*, vol. 13, pp. 99-108, 1997.
3. P.B. Van Verth, "A System for Automatically Grading Program Quality," *SUNY (Buffalo) Technical Report*, 1985 .
4. G. Michaelson, "Automatic analysis of functional program style," *Australian Software Engineering Conference*, vol. 13, pp. 38-46, Melbourne, Australia, 1996.
5. P. Naur and B. Randell, *Software Engineering*, 1968. NATO
6. M. Halstead, *Elements of Software Science,* Elsevier Scientific Publishing Co., 1977.
7. J. McCall, P. Richards, and G. Walters, *Factors in Software Quality, 3 Vols.*, 1977. NTIS AD-A049-014,015,055
8. J.A.W. Faidhi, "The complexity analysis of Pascal programs and the application to a university teaching environment," *PhD Thesis*, University of Brunel, 1986.
9. A. M. Zin and E. Foxley, "Automatic Program Quality Assessment System," *Proceedings of the IFIP Conference on Software Quality*, March 1991. S P University, Vidyanagar, INDIA
10. S. Hung, L. Kwok, and A. Chung, "New Metrics for Automated Programming Assessment," *IFIP Transactions A-Computer Science and Technology*, vol. 40,

pp. 233-243, 1993.

11. R.S. Pressman, *Software Engineering*, McGraw-Hill Company Europe, 1992.
12. N.F. Schneidewind, "Standards," *Computer*, April 1993.
13. S.S. Yau and J.S. Collofello, "Some Stability Measures for Software Maintenance," *IEEE Transactions on Software Engineering*, vol. 6, no. 6, pp. 545-552, November 1980.
14. R.S. Burgess, *An Introduction to Program design using JSP*, Hutchinson & Co. Publisher Ltd, 1984.
15. A.M. Zin and E. Foxley, "Analyse - An automatic program assessment system," *Malaysian Journal of Computer Science*, vol. 7, p. 123, 1994.
16. R. Whitty, "Structural Metrics for Z Specifications," *Fourth Annual ZUM*, Rewley House, Oxford, UK, 15 December 1989.
17. S. Henry and D. Kafura, "On the relationship among three software metrics," *Perform. Eval. Rev.* , no. 10, 1 , pp. 81-88, Spring 1981.
18. T.J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308-320, December 1976.
19. K.A. Redish and W.F. Smyth, "Evaluating Measures of Program Quality," *The Computer Journal*, vol. 30, no. 3, 1987.
20. M.J. Rees, "Automatic Assessment Aid for Pascal Programs," *SIGPLAN Notices*, vol. 17 , no. 10, pp. 33-42, October 1982.
21. S.H. Valentine, "The programming language Z--," *Information and Software Technology*, vol. 37, no. 5-6, pp. 293-301, 1995.
22. P.A.V. Hall, "Towards Testing with Respect to Formal Specifications," *Proc. of Second IEE/BCS Conference: Software Engineering 88*, pp. 159-163, London, 1988.
23. P.W. Oman and C.R. Cook, "A Paradigm for Programming Style Research," *SIGPLAN Notices*, vol. 23, no. 12, pp. 69-79.

24. R. Barden, S. Stepney, and D. Cooper, *Z in Practice,* Prentice Hall, 1994.

25. J.B. Wordsworth, "Education in formal methods for software engineering," *Information and Software Technology*, vol. 29, Jan-Feb 1987.

26. V.R. Basili, "Tutorial on Models and Methods for Software Management and Engineering. ," *IEEE Computer Society Press*, Los Alamitos, California, 1980.

27. J.K. Kearney, R.L. Sedlmeyer, W.B. Thompson, and M.A. Gray, "Software Complexity Measurement," *Communications of the ACM*, vol. 29, no. 11, pp. 1044-1050, September 1986.

28. W. Curtis, "Management and Experimentation in Software Engineering," *Proceeding of the IEEE*, vol. 68, no. 9, September 1980.

29. V.R. Basili, R.W.Jr. Selby, and Philips T., "Metric analysis and validation across Fortran projects," *IEEE transaction Software Engineering SE-9*, pp. 652-663, Nov 1983.

30. A.M. Zin, *ZFDSS: A Formal Development Support System based on the Liberal Approach*, 1994. PhD Thesis, University of Nottingham, UK

31. B. Fields and M. Elovang-Goransson, "A VDM Case Study in mural," *IEEE Trans. Software Eng.*, vol. 18, no. 4, pp. 279-295, Apr. 1992.

32. J.A. Goguen, "Parameterized Programming," *IEEE Transactions on Software Engineering*, vol. 10, no. 5, pp. 528-543, 1984.

33. R. Balzer and N. Goldman, "Principles of Good Software Specification and their Implications for Specification Languages," *Proceedings of IEEE Conference on Specifications of Reliable Software*, pp. 58-67, Cambridge, Mass., 1979.

34. J.C.S. do Prado Leite and P.A. Freeman, "Requirements Validation Through Viewpoint Resolution," *IEEE Transactions on Software Engineering*, vol. 17, no. 12, pp. 1253-1269 , December 1991.

35. R.A. Kemmerer, "Testing Formal Specifications to Detect Design Errors," *IEEE Transactions on Software Engineering*, vol. 11, no. 1, pp. 32-43, January

1985.

36. P. Jalote, "Testing the Completeness of Specification," *IEEE Transactions on Software Engineering*, vol. 15, no. 5, pp. 526-531, May 1989.

37. N.P.H. Haigh, "Providing tool support for Z," *Software Tools: Improving Applications*, pp. 185-191, June 1987.

38. D. Garlan, "Making formal methods education effective for professional software engineers," *Information and Software Technology*, vol. 37, no. 5-6, pp. 261-268, 1995.

39. C. Parker, "Z Tools Catalogue," ZIP/BAe/90/020, Software Technology Dept, British Aerospace, 10 May 1991.

40. P. Steggles and J. Hulance, *Z Tools Survey,* Imperial Software Technology Ltd & Formal System (Europe) Ltd, June 1994.

41. S. Benford, E. Burke, E. Foxley, N. Gutteridge, and A.M. Zin, "CEILIDIH: A Course Administration and Marking System," *Proceedings of International Conference in Computer based Learning in Science*, Vienna, 1993.

42. A. M. Zin and E. Foxley, "The Oracle Program," *LTR Report*, Computer Science Dept, Nottingham University, 1992.

43. E. Foxley and A.M. Zin, *Zpp - A Troff Preprocessor for Typesetting Z Specifications*, 1990. Nottingham University Computer Science

44. K.A. Redish and W.F. Smyth, "Program Style Analysis: A Natural By-Product of Program Compilation," *Communications of the ACM*, vol. 29, no. 2, pp. 126-133, February 1986.

Chapter 3
Automatic Z Specification Assessment System

3.1. Introduction

In this chapter, we will describe the implementation of the Automatic Z Specification Assessment System (AZAS), which is designed for assessing the quality of Z specifications written by beginners. The system is now being used as one of the components in a course management system called Ceilidh.

The system was implemented under the UNIX operating system. It was written using the C++ and C languages, and UNIX facilities such as `awk`, `sed` and shell programming. The system consists of four different main programs; each program is used to assess different factors of a Z specification's quality. The Prolog system `sicstus` is used to support the process of dynamic marking. The system relies upon the type and syntax checker for Z specifications, `zc`,[1] as well as the Z to Prolog translator, `zp`.[1] The type checker and the translator have been developed by Zin[1] and are used mainly in the University of Nottingham.

The structure of AZAS can be split into four important tools that are specifically designed for marking different aspects of quality; typographic arrangement of a Z specification, complexity of a Z specification, its static correctness and its dynamic correctness. A brief diagram of the AZAS structure is shown as in Figure 3-1.

Besides the student's solution, AZAS also needs a teacher's model answer, as well as other particulars to be used in the assessment process. Each of the tools will produce their own marks which have been scaled to a percentage, and these marks will then be totaled up according to weight to make the overall marks. The student can be informed of their result in several ways, depending on the teacher's intention.

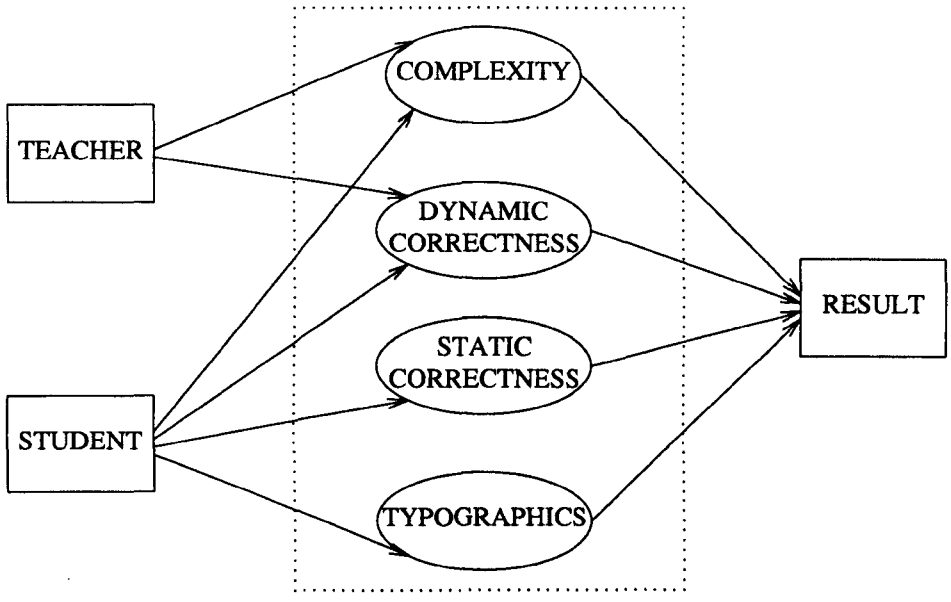


Figure 3-1 : Framework of AZAS

This chapter is divided into four main sections: maintainability which involves typographic and complexity, and correctness which involves static correctness and dynamic correctness. Every section will start by describing related research done by other researchers, followed by an explaination of how the factors are being assessed and finished by describing the program involved. This chapter ends by showing the output of the system.

A brief description of the system with an initial experimental result was presented at the PROGRESS '98 Conference and published in the proceedings.[2] The content of this chapter has been accepted to be published in the Journal of Computing in Higher Education.[3]

3.2. Maintainability

The system measures a Z specification's maintainability by analysing the text based on two aspects; typographic arrangement and structural contents, i.e complexity.

3.2.1. Typographics

The following section is a discussion about some of the selected factors.

The first factor that will be discussed here is an informal description. It is obvious that informal description which is written in natural language, e.g. the English language, is very important in any specification as it describes what is specified by the Z component. It helps the reader to interpret the formal description.[4] A Z specification is composed of many schemas. The informal description which explains the significance of the formal mathematics therefore plays a role to act as a link between those schemas.[5] However, the amount of English description and Z description in a specification should be balanced so that it helps the reader to understand the specification more efficiently. A study by Jones[6] shows that a number of specifications lack informal description. The underlying issue here is what percentage the English description should form in any specification. Most articles mention the importance of the English description, but none discuss the percentage. Further, spelling errors should not happen in the English description. The system uses the Unix System Vocabulary to check the spelling. However, we must consider a few types of words that might not be in the Unix system vocabulary, which will be stated in the problem statement.

The second factor is the style convention. It is sensible to follow an agreed convention for choosing identifiers in Z specifications. This makes the specification more readable. We follow the convention recommended by Gravell[7] and MacDonald,[8] which concerns Case. Conventional style for Case in Z is as below:

- Schema components, global variables and local variables should be written in lower case letters.
- Schema names should be written in mixed upper and lower case letters.
- Types should be written in upper case letters.

- Constants should be written in upper case letters.

Further convention styles are:

- "?" and "!" are used at the end of input and output variables, respectively.
- The symbol Δ is used in front of a schema name as an indicator that the schema is in association with a state change and the symbol Ξ for an unchanged value of a state.
- If the total operation is called *Op*, then the correct part can be called *Op_OK* and the error cases can be called *Op_Error_1*, *Op_Error_2* and so on. So the total operation would be:

$$Op \triangleq Op\_OK \lor Op\_Error\_1 \lor Op\_Error\_2 \lor \cdots$$

However our system does not take these styles into account because the style is inherited by the support tools (i.e editor, type and syntax checker).

The third concern is the length of the identifiers. It has been stated that meaningful identifiers improve comprehension.[9] However, an identifier name that is too short can lead to misunderstanding and confusion whilst a complete name will use too many spaces and provide a potential error.[10] One-letter names for variables are not advised to be used, unless they are used locally.[8] A suitable length for the identifier is therefore needed to cater for both problems. Most of the references stated that an acceptable identifier is a string of reasonable length. However they mention nothing about the value or range of the acceptable lengths.

The fourth matter is schema length. Schemas should be simple so that the whole specification is easier to understand and errors are more likely to be spotted.[8] The unit that we used in counting the schema length is a line. For example, if the schema takes 5 lines, the length of the schema is 5.

Dealing with long quantified expressions can be confusing. Thus, it can be useful to follow indentation, with the key symbols indented by the same amount.[8] In our system, indentation is captured when any lines end with an operator. Otherwise, the

line should not be indented.

As we have discussed earlier, there are many more factors that contribute to composing a good quality specification layout. At the moment our system chooses only factors that can be calculated and produce a numeric value; we claim that this is satisfactory for assessing Z specification written by beginners of Z. Nevertheless other factors could be researched, and if they are taken into consideration we might have a potentially intelligent system that could assist Z practitioners to write high quality Z specifications. The factors selected for the typographic arrangement in our system are:

- % of blank lines
- average schema length
- % of schemas which have a good length
- average variable name length
- % of variable names which have a good length
- % adherence to style conventions (to be agreed)
- correctness of indentation
- juxtaposition of English text and Z schema
- % of spelling errors (explanatory)

The typographic style is concerned with the specification presentation, it is independent of the particular requirements statement and model specification being assessed. Table 3-1 shows the range of values that we choose to be the indicator of quality in typographic aspect. This range were chosen after studying several examples of Z specification from[11-13] . Each of the factors are marked relative to that range of values. These values can easily be reset. Table 3-2 shows the formula that the system uses to calculate every factor, with an explanation of the formula in the last column.

Factor	L	S	F	H
% English description	6	11	60	65
% of blank lines	-3	0	25	28
% of wrong spell	-2	0	20	22
% of conventional	78	80	100	102
Average of length identifiers	1	3	25	27
% of variable names which have a good length	78	80	100	102
Average schema length	3	5	19	21
% of schemas which have a good length	78	80	100	102
% of Z Description	24	30	85	91
% of Good indented	78	80	100	102
% of bad indented	-2	0	20	22

Table 3-1 : Typographics Quality Parameters

Factor	formula	explanation
% English description	(m/n) x 100	m is a number of English description lines and n is a total number of lines
% of blank lines	(m/n) x 100	m is a number of blank lines (all blank lines from start to end of file) and n is a total number of lines. Please note that blank lines at the beginning and end of files are not counted.
% of wrong spell	% of wrong spell	m is a number of words that spelled wrongly and n is a total number of words in English description.
% of conventional	(m/n) x 100	m is a number of identifiers that satisfy the rules and n is a total number of identifiers.
Average of length identifiers	m/n	m is a total length of all identifiers and n is a total number of identifiers.
% of variable names which have a good length	(m/n) x 100	m is a number of words with good length and n is a total number of all words.
Average schema length	m/n	m is a total of lines that used to write a Z schema and n is a total number of Z schemas.
% of schemas which have a good length	(m/n) x 100	m is a number of schemas with good length and n is a total number of all Z schemas.
% of Z Description	(m/n) x 100	m is a number of Z decsription lines and n is a total number of lines.
% of Good indented	(m/n) x 100	m is a number of lines with good indented and n is a total number of lines.
% of bad indented	(m/n) x 100	m is a number of lines with bad indented and n is a total number of lines.

Table 3-2 : Typographics Calculation

Scoring for typographics

The awarding of scores for each aspect of typographic style is done by using a technique proposed by Rees.[14] The teacher is required to provide the maximum mark for each factor. Marks will be awarded for the percentage of factors that falls within a certain range of values. The system keeps this detail in a special file named *model.tv*. The format of the file is that each line starting with 4 characters represents the short-form of the factor's name, followed by a maximum mark, and four values defining the five ranges of marks awarded. Below (Figure 3-2) is an example of a typical file *model.tv*.

Factors	Maximum Mark	L	F	S	H
%ENG	5	6	11	60	65
%BLL	5	-3	0	25	28
—	—	—	—	—	—
—	—	—	—	—	—
—	—	—	—	—	—

Figure 3-2 : Example of file *model.tv*

The formula to calculate the mark for each type of error is

if $F \leq X \leq S$ then award *Maximum Mark*

if $L \leq X < F$ then award $\frac{X-L}{F-L}$ *Maximum Mark*

if $S < X \leq H$ then award $\frac{H-X}{H-S}$ *Maximum Mark*

if $X < L$ or $X > H$ then award no mark

where

X is the value of any factors (calculated in Table 3-2).

L, F, S and H are values represent the five ranges of marks awarded (see Table 3-1).

Example. Assume that the above is detailed in the file *model.tv*. Suppose that 63% of the student's specification consists of English description. The student will then be awarded 2 out of 5 marks for that factor. The total of the maximum marks for every factor will then be scaled to a percentage.

Program description : `typog_z`

The program is written in C++. It takes a student solution as an input, and analyses it line by line. To check for spelling errors, the program uses the spell checker in the Unix system.

The syntax of the command for `typog_z` is :

```
typog_z  [-v0 | -v1 | -v2 | -v3 ] [-f0 | -f1 | -f2 | -f3 ]
         [-w file1] file2
```

where file1 is a weights file and file2 is a Z specification document with extension .z. The explanation of the flags is as follows:

-w : use given file weight

-v0 | -v1 | -v2 | -v3 : level of verbosity (Numeric)

(v0 and v1 show only the final score, v2 shows the score for every factor and v3 shows the score for every factor as well as the maximum mark for every factor)

-f0 | -f1 | -f2 | -f3 : level of verbosity (Feedback in English)

(f0 shows only the final score, f1 shows the factors that lost most marks, f2 shows all the factors that lost marks and f3 comments on marks for all the factors by using word either excellent, well, moderate, fair or bad)

3.2.2. Complexity

The factors of Z specification structure in our domain are mostly taken from the ideas of Halstead Software Science, that is, counting the lexical entities of Z. This category includes the analysis of the following factors.

- operators
- number of operators per schema
- mathematical toolkit operations
- complexity of predicate expressions
- complexity of schema algebra expressions
- number of schemas
- number of operation schemas
- number of distinct identifiers
- total number of variables
- number of distinct variables
- number of observation schemas
- schema inclusions
- depth of schema inclusion
- number of variables per schema
- basic/free type definitions
- average length of natural language texts
- the use of schema expressions

Each one is marked relative to the corresponding counts for the model specification, and we will again assume the scoring technique shown in Figure 2-1. In a typical factor we might allow the student full marks for being within a factor of 2 (from 50% to 200%) of the corresponding metric for the model solution.

In our system, not all the above factors are calculated. Table 3-3 shows the factors that are chosen (pragmatically) to be included in the system.

Factor	Technical Description
Number of distinct identifiers	This includes schema names, schema components, given types and constants. Please note that any identifiers with 'prime' symbol is not counted.
Number of schemas	This involves schema definition and schema calculus.
Number of operation schemas	This involves schema which can change the state.
Number of observation schemas	This involves schema which cannot change the state.
Average of schema inclusion	This means the number of schema inclusion per schema.
Average depth of schema inclusion	This means the depth of schema inclusion per schema. A schema which does not include any schema in it, has a depth of 0. The depth of schema which include other schemas is, the number of included schemas + the depth of every included schema.
Number of total variables	This involves global variables, schema variables and local variables . Note: variables with 'prime' symbol is not counted.
Number of distinct variables	This involves only distinct variables is counted.
Number of total constants	This involves abbreviation, user-defined symbol and functional and relational operator (infix, prefix and postfix are not implemented).
Number of distinct constants	This involves only distinct constant is counted.
Number of total given types	This involves basic type, free type and generic type.
Number of distinct given types	This involves only distinct given types is counted.
Number of total basic built types	Basic built type is any types except given type and composite type. For example N.
Number of total composite types	This involves distinct variables that associated with the composite types.
Average composite type length	This means the average number of composite types been used per distinct variables type.
Number of predicates	This involves the schema predicates declaration and constraints for global variables. One predicate means one line in predicate part.
Number of operators	This involves operators which occurs in the schema predicates and in constraints expression for global variables. Operators under schema calculus are not counted.
Average operators per schema	This means the average number of operators per schema, excluding schema calculus.
Average operators per predicate	This means the average number of operators per predicate.
Average predicates per schema	This means the average number of predicates per schema, excluding schema calculus.
Average variables per schema	-

Table 3-3 : Complexity Calculation

Scoring for complexity

The student specification metrics must be within certain factors of those of the model solution for each metric set by the teacher. The marks and parameters for each of the factors are kept in file *model.metric*. Each line contains 4 characters which represent a code for the factors, followed by a maximum mark, next two values representing L and F, then a metric from the model solution, with the last two values representing S and H. Below (Figure 3-3) is an example of a typical file *model.metric*.

Factors	Maximum Mark	L	F	Model	S	H
NDID	5	8	16	32	64	96
NSCH	5	0.50	1	2	4	6
-	-	-	-	-	-	
-	-	-	-	-	-	
-	-	-	-	-	-	

Figure 3-3 : Example of file *model.metric*

The formula to calculate the mark for each type of factor is similar to the typographics formula (i.e using the Rees model).

Example. Assume that the above details are in the file *model.metric*. If the student has 80 different identifiers and the model has 32, the student gets 2.5 out of 5 marks for that factor. Similar to typographics marking, the total marks of every factor will be scaled to 100%.

Program description : `complex_z`

`zc` is used to support the program in obtaining those metrics. The program analyses files which consist of a conceptual model of a specification which is generated by `zc`. Please note that complexity marking cannot be done, unless the specification has been succesfully compiled by `zc`.

The relevant weights file (that is a file with extension .metric) from a model solution should first be generated before invoking the main process. The command to

generate this file is:

```
complex_z -mm file
```

where file is a Z specification document (model solution) with extension .z.

The syntax of the command for `complex_z` is :

```
complex_z [-v0 | -v1 | -v2 | -v3 ] [-f0 | -f1 | -f2 | -f3 ]
          -w file1 file2
```

where file1 is a weights file and file2 is a Z specification documents with extension .z.
The explanation of the flags is as follows:

-w : flag weight, use following argument as file weight
-v0 | -v1 | -v2 | -v3 : level of verbosity (Numeric)
(v0 and v1 show only the final score, v2 shows the score for every factor and v3 shows the score for every factor as well as the maximum mark for every factor)
-f0 | -f1 | -f2 | -f3 : level of verbosity (Feedback in English)
(f0 shows only the final score, f1 shows the factors that lost most marks, f2 shows all the factors that lost marks and f3 comments on marks for all the factors by using word either excellent, well, moderate, fair or bad)

3.3. Specification correctness

As explained in chapter 2, the system measures the correctness of a specification using two steps.

1. The specification's static correctness is checked.
2. Next, a number of sets of test data are run against the specification.

3.3.1. Static correctness

Static analysis involves the examination of the text of the specification without 'execution'. The specification structure and syntax are inspected to highlight 'static' errors and produce statistical information for the specifier.[15] In order to assess static correctness, we rely partly on `zc`. This compiler will comment on Z specifications as part of the compilation process. The score for static analysis will be based on the outcome of applying `zc` and the occurrence of other static problems in the specification. This checks for Z syntax, for correct types in all expressions, and for the validity of all use of identifiers.

The Z compiler analyses a Z specification document written in Z-roff format in order to build a Z specification conceptual model. The conceptual model for a Z specification defines the software objects and relationships between these objects at a selected level of abstraction.[1] Another task of the compiler is to perform error reporting. The compiler classifies the errors into three types, lexical errors, syntax errors and semantic errors.

Lexical errors are concerned with the validation of tokens. For example, variable *name@* is an invalid token in a Z specification.

Syntax errors occur when the specification does not follow the Z syntax rules as specified by Spivey.[16]

Semantic errors involve scope errors and type errors. An example of scope analysis is that all variables declared in a schema have schema scope. An error will therefore occur if a schema uses a variable which is declared in another schema. An example of type checking is that the compiler follows *predicate1* <*rel*> *predicate2* syntax for an infix relation. The type of *predicate1* and *predicate2* must conform with the types required by the relation <*rel*>. If the relation is $\subseteq$, then *predicate1* and *predicate2* should be the same type of set.

Scoring for static correctness

Static correctness is measured by counting the errors detected when we execute `zc`. For each of the three types of error summarised above (lexical, syntax, semantic), we give a maximum mark. A mark is then deducted from the maximum mark for every occurrence of an error of that type. In addition to a maximum mark, a weight for every type of error is also given. The details of the maximum mark and the weight for every error are stored in a file *model.sv*. This detail can be changed by the teacher to reflect different requirements or priorities. Below (Figure 3-4) is an example of a typical *model.sv* file.

Type of Error	Maximum Mark	Weight
Lexical	5	1
Syntax	5	3
Semantic	5	2

Figure 3-4 : Example of file *model.sv*

The formula to calculate the mark for each type of error is

(Maximum mark - Number of errors) * Weight

subject to a non-negative result.

Example. Let us say that in one particular exercise the student had 1 lexical error, 2 syntax errors and 4 semantic errors. Using the detail in the above *model.sv* file as an example, the student score is calculated as below (Figure 3-5):

for lexical	(5 - 1) x 1	=	4
for syntax	(5 - 2) x 3	=	9
for semantic	(5 - 4) x 2	=	2
Total lexical score		=	15

Figure 3-5 : Calculation for static correctness mark

The total score is 15 out of 30, which is 50%. If the student has errors equal to or more than maximum mark, then 0 marks will be given to the respective type of error.

If the student has no errors then she or he will be awarded full marks of 100%.

Program description : `statcorr_z`

By modifying `zc`, instead of producing error messages, it produces the number of errors of each type. The marks for each error are either taken from default values from data initialization built in the `statcorr_z` program, or can be supplied from a file *model.sv*.

The syntax of the command for `statcorr_z` is :

```
statcorr_z [-w file1] [-v0|-v1|-v2|-v3] [-f0|-f1|-f2|-f3]
           file2
```

where file1 is a weights file and file2 is a Z specification document with extension .z. The explanation of the flags is as follows:

-w : use given weights file
-v0 | -v1 | -v2 | -v3 : level of verbosity (Numeric)
(v0 and v1 show only the final score, v2 shows the score for every factor and v3 shows the score for every factor as well as the maximum mark for every factor)
-f0 | -f1 | -f2 | -f3 : level of verbosity (Feedback in English)
(f0 shows only the final score, f1 shows the factors that lost most marks, f2 shows all the factors that lost marks and f3 comments on marks for all the factors by using word either excellent, well, moderate, fair or bad)

3.3.2. Dynamic correctness

In order to check the degree to which the specification fulfills the user requirements, the system uses a testing technique. The *execution* is done by animating the specification. The main aim of animation is two fold. Firstly, we wish to uncover any inconsistency or incompleteness in the specification. Secondly, we must ensure

that the specification faithfully reflects the 'customer' requirements.

A Z specification consists of schema paragraphs. Some are *state schemas* defining system invariants. Others are *operation schemas* defining the relation between the states before and after a given operation. The definition represented by the schema is written as a number of predicate formulae. To test that the schema is defining the behaviour of the system correctly, we invoke the schema with a chosen set of values of the system variables; these may include global pre- and post- variables, as well as input and output variables. If the schema is successfully invoked, we continue by analysing the values of the rest of the system variables. A measure of the system's behaviour is determined by the expected values of the system variables. The complete analysis of the schema will involve a number of tests; some marks will be awarded for each test, and the total mark represents (in a certain sense) the overall success of the schema.

We will consider the example of *telephone book.* The example involves storing the names and telephone numbers of a group of people; the state schema is named *TelephoneBook,* the schema representing the addition of a new entry is *AddTelephone,* and the schema representing the retrieval of a named person's telephone number is *FindTelephone.*

Our assessment system might invoke the student's copy of an event schema such as *AddTelephone* with particular values for the initial state and for the input variables (this will be referred to as a *test case*) and check the final state and output values with tests such as (Figure 3-6)

25 marks:	$eric \in known'$
5 marks:	$known' = \cdots$
10 marks:	$telephone' = \cdots$
5 marks:	$result! = \cdots$

Figure 3-6 : Example of final state and output analysis

where *eric* is the new name being added, *known* is the set of all people whose names

are known, *telephone* is a partial function which maps names to numbers, and *result* is some result message indication successful conclusion. By giving this example, we do not mean to abandon the symmetrical feature of the input and output in the Z language. The implementation of this feature in the system will be discussed in chapter 4.

Assessing state schema

There is a question as to how we should test the student's state schema. We need to set up initial valid and invalid states, and check that the student's state schema reports correctly. In one sense this is a static test, but we implement it using animation.

Assessment using animation

Three entities are involved in each animation test in the process; these are the specification, the test case and the test weight. The test case is a file which is written by the teacher to set up a data state for the schema. It is written in the format

$$
\begin{aligned}
&\mathit{SchemaName} \\
&\mathit{variable}_1 = \mathit{value}_1 \\
&\mathit{variable}_2 = \mathit{value}_2 \\
&\quad . \ . \ .
\end{aligned}
$$

where

SchemaName is the name of the schema that we want to test,
$\mathit{variable}_1$ etc will be the name of an input, output or pre- or post- state variables and
value_1 etc is the value we wish that variable to take.

The file defining the weight to give to each test specifies the predicates we wish to test, and the mark to be awarded for each successful test. The format of a test weight file is as below:

SchemaName

$mark_1$: $predicate_1$

$mark_2$: $predicate_2$

. . .

where

SchemaName is name of the schema that we have tested and

$mark_1$ is the mark we want to give if $predicate_1$ is *true*.

To make the specification *executable,* we translate the specification into Prolog. Not all specifications can be made executable, suitable choice of exercise needs to be made by teacher. An intuitive motivation for this is that mathematical logic forms the foundation of both Z and Prolog. We use `zp`[1] to translate Z specifications into Prolog code. The test cases and test weights will also be translated into Prolog, this time using `scrptz` and `markz` respectively (see chapter 4).

We then run the specification (which is now Prolog code) by adding several queries, and analysing the values of output variables.

Example assessing state schema : *TelephoneBook*

We will show a simple example of how we can apply this technique to grade the degree to which a state schema fulfills a specification. We will use the above example of a system which records people's telephone numbers. A correct state schema for the system is as follows.

TelephoneBook

$known : \mathbb{P}\ NAME$

$telephone : NAME \pfun PHONE\_NUMBER$

$known = \mathbf{dom}\ telephone$

We would wish to check that a schema supplied by a student has a behaviour

equivalent to this.

The state schema's predicates given in the predicate part should hold in all valid cases, and the conjunction of the predicates should fail for invalid data. To test a state schema, we actually test whether the relations in its predicate behave as required. We give certain values to its variables, and then invoke the schema.

In the above schema, there are two state variables, *known* and *telephone*. *known* is a set of *NAME*s, whilst *telephone* is a partial function from *NAME*s to *PHONE_NUM*s.

The predicate part of the schema specifies that the domain of *telephone* is equal to the value of *known*. We are interested when checking any student schema to see whether the relation

$$known = \mathbf{dom}\ telephone$$

is correctly *true* or *false* when applied to certain values of the variables supplied to or defined by the student schema. We can give at least two sets of variable values, one in which the predicates of the state schema should hold, and another in which the conjunction of the predicates fails.

An example of a test case which should satisfy the predicate is

$$telephone = \{(john,\ 9790917),\ (mary,\ 9788777)\}$$
$$known = \{john,\ mary\}$$

The sets

$$telephone = \{(john,\ 9790917),\ (mary,\ 9788777)\}$$
$$known = \{mary\}$$

and

$$telephone = \{(john,\ 9790917),\ (mary,\ 9788777)\}$$
$$known = \{eric,\ john,\ mary\}$$

should cause the state schema to deliver *false*. More serious possible student errors would arise from type errors such as

$telephone = \{(john, 9790917, mary), (mary, 9788777, john)\}$
$known = \{eric, john, mary\}$

or

$telephone = \{(john, 9790917), (mary, 9788777)\}$
$known = \{9790917, 9788777\}$

Accompanying each test case is a progressive marking scheme that tests the extent to which the student's schema fulfills the required conditions. We may require with a valid initial data set that

10 marks if the relation '*known* = **dom** *telephone*' is *true*

We must also test the correctness of the relation by giving an invalid initial data state, in which the state schema should deliver the result *false*. For example, the data set

$telephone = \{(john, 9790917), (mary, 9788777)\}$
$known = \{john\}$

should have a marking scheme

10 marks if the relation '*known* = **dom** *telephone*' is *false*

```
.ZS TelephoneBook
known : POWER NAME
telephone : NAME PARTFUN PHONE_NUM
.-
known = DOM telephone
.ZE
```

Figure 3-7 : Z specification in Z-roff format

Figure 3-7 shows the Z-roff format of the model specification as supplied by the teacher, Figure 3-8 shows the file defining the data state for the first test case and Figure 3-9 shows the file defining the marking for this data state (in this case "10 marks is given if the schema is TRUE").

TelephoneBook
telephone = { (john, 9790917) , (mary, 9788777) }
known = { john, mary }

Figure 3-8 : First test data state

TelephoneBook
10 : TRUE

Figure 3-9 : First test data marking

The above example shows how we assess the correctness of a state schema. We will also wish to assess schema which represent a change to the data state, an *operation schema* , and of schema for reporting on the data state, an *observation schema* .

Examples assessing operation schema : *AddTelephone*

The following example still uses the same *telephone book* problem, with the above state schema. In the problem, there is an operation schema, *AddTelephone*. This takes two inputs *name* ? and *phone* ? representing the name and telephone number of a new entry, thus changing the data state, but producing no output. The new state variable *telephone'* will be updated by adding a new relation that is (*name* ?, *phone* ?). The expected specification of the schema is

AddTelephone

$\Delta TelephoneBook$
$name? : NAME$
$phone? : PHONE\_NUM$

$telephone' = telephone \cup \{name? \mapsto phone?\}$
$known' = known \cup \{name?\}$

Critical errors such as the non-existence of input variables or the number of input variables more than expected are ignored for the time being. The topic will be discussed in chapter 4.

Assume that we want to add a person's name, *John* and his telephone number is 9790917 into the *TelephoneBook* database. We also want to initialise the value of the data state *known* and *telephone* as empty sets. Hence the state of the system after the operation for *known'* and *telephone'* are $\{John\}$ and $\{(John, 9790917)\}$, respectively.

We can now design our test cases for this operation. For the operation schema the test case is

AddTelephone

$name? = john$

$phone? = 9790917$

$known = \{\}$

$telephone = \{\}$

Our mark scheme expressed informally would be

-if the conjunction of all predicates in the student's schema *AddTelephone* that is

$telephone' = telephone \cup \{name? \mapsto phone?\} \wedge known' = known \cup \{name?\}$

is *true*, give 10 marks

-if *known'* contains an element *John*, give 5 marks

-if *telephone'* contains a relation (*John*, 9790917), give 5 marks

These sentences have to be transformed into special notation, and the file representing the marking for this test is

AddTelephone

10 : *true*

5 : $john \in known'$

5 : $(john,\ 9790917) \in telephone'$

If the schema should be TRUE when it fails, then the system will award a mark according to the percentage of predicates which contribute to the corrrectness. For example, the schema *AddTelephone* consists of a total of three predicates (one predicate is from the included schema *TelephoneBook*). The predicates are as follows:

$known = \mathbf{dom}\ telephone$

$telephone' = telephone \cup \{name? \mapsto phone?\}$

$known' = known \cup \{name?\}$

Let us say that the second predicate of the schema causes an error which fails the whole schema. The analysis could not be continued due to the nature of Prolog implementation. This means that one third of the predicates has been tested working (i.e the $known = \mathbf{dom}\ telephone$). Therefore 33.33% from the amount of the marks when the schema is true, will be awarded by the system. In this case, 33.33% out of 10, that is 3.33 marks will be awarded to the student.

This situation, along with a technique to design a test case and to plan a marking scheme for the purpose of dynamic marking is presented in detail in chapter 4.

Scoring for dynamic correctness

Each of the test case files with their corresponding progressive mark file will be run against the student specification. Each test case will have an overall weight. The output from the process is either the total of the progressive marks multiplied by their

weighting factors, or a negative value showing that an error has occurred (the student schema may have syntax or type errors). This mark is then scaled to an integer percentage.

If there are six test cases, the six overall test weights are given in the file *model.dv*, typically (Figure 3-10)

Test number	Title of the test	Test Weight
1	TelephoneBook	4
2	AddTelephone, general test	8
3	AddTelephone, add to empty database	6
4	AddTelephone, add existing user	6
5	FindTelephone, general test	6
6	FindTelephone, user not found	4

Figure 3-10 : Example of file *model.dv*

Assume that in one particular exercise, the progressive marks awarded to the student for the six tests are 40%, 80%, 20%, 30%, 60% and 10% respectively. These marks will be totaled using the given weighting factors and scaled to 100%. The overall cumulative mark will be

$$(40*4 + 80*8 + 20*6 + 30*6 + 60*6 + 10*4) / (4 + 8 + 6 + 6 + 6 + 4) * 100$$

Negative marks act only as indicators to the type of failure, and are always treated in calculations as a mark of zero. The indicators are

-1 : Runtime prolog error.

-2 : Format error.

-3 : Prolog loading error.

Program description

The process in carrying out the dynamic marking is discussed under the issue of testing the correctness of a specification, presented in the next chapter.

3.4. The overall system

Basically, the system will receive as input

the student specification,
a set of test data and,
a marking scheme.

The test data would be provided by the teacher in an educational context. The marking scheme will also be provided by the teacher. Typically, the system marks the specification by first analysing its layout and then its static correctness. If there are no errors in static correctness, the system will continue the marking process, to analyse the specification's complexity, and lastly to animate the specification against the test data and award marks to represent its dynamic correctness. However, it is not necessary to have four factors in every analysis. It depends on the marking scheme set by the teacher. In a simple question, the focus is usually on correctness; maintainability is not so important.

The output of the system is designed so that it is helpful to the student. The system has three ways of presenting the output. The first is by showing details of the score and the mark for every factor. The second is verbal grading for every factor. Lastly, only the overall score is shown. The style is chosen by setting the respective flag when we run the program. Figure 3-11 is an example of output from the static correctness marking.

Static Correctness Marks			
ERROR TYPE	NUMBER OF ERRORS	MARK	OUT OF
Semantic	0	10	10
Lexical	0	10	10
Syntax	0	10	10
Total Static Correctness			100.00%

Figure 3-11 : Example of output from static correctness marking

The output is separated into four columns. The first column shows the type of error; the second column shows the number of errors that are produced from the student solution; the third column displays the marks that are awarded to the student with respect to the occurrences of errors (and it is scaled to the weight); and the last column shows the weights of the errors. The overall mark is shown in the last row. It is scaled to a percentage.

Figure 3-12 shows an example of output from animation marking.

Dynamic Correctness Marks			
TEST	SCORE	MARK	OUT OF
Simple Test	50	10	10
Harder Test	15	19	25
Difficult Test	25	13	25
Overall Dynamic Correctness mark			70.00%

Figure 3-12 : Example of output from dynamic marking

The output is divided into four columns. The first one describes the type of the test; the second, shows the unscaled mark that is awarded to the student for a respective test; the third column presents a mark which has been scaled to the weight; and the last column shows the weights of the tests. The total mark which is scaled as a percentage is shown at the bottom. The same output style is used for typographics where the output is divided into four colums, where the only difference is at the first column, and we now show the type of factor. This is shown in Figure 3-13.

Typographic Marks			
ITEM	SCORE	MARK	OUT OF
% of English Description	49.06	5.00	5.00
% of blank lines	0.00	5.00	5.00
% of wrong spell	0.90	5.00	5.00
% of conventional	100.00	5.00	5.00
Average Identifier length	6.80	5.00	5.00
% identifier with good length	10.00	0.00	5.00
Average schema length	5.50	0.50	5.00
% schema with good length	0.00	0.00	5.00
% of Z description	50.94	5.00	5.00
% of good indent	100.00	5.00	5.00
% of bad indent	0.00	5.00	5.00
Total Typographics			73.64%

Figure 3-13 : Example of output from typographics marking

Finally, Figure 3-14 shows an example of output from complexity marking. The output is similar to typographics with an extra column which shows the metric from the model specification.

Complexity Marks				
ITEM	MODEL	SCORE	MARK	OUT OF
Number of distinct identifier	21.00	21.00	5.00	5.00
Number of Schema	3.00	2.00	5.00	5.00
Number of Total Variable	12.00	12.00	5.00	5.00
Number of Distinct Variable	11.00	12.00	5.00	5.00
Number of Total Constant	0.00	0.00	5.00	5.00
Number of Distinct Constant	0.00	0.00	5.00	5.00
Number of Total Given Type	3.00	2.00	5.00	5.00
Number of Distinct Given Type	7.00	7.00	5.00	5.00
Total Basic Built Type	0.00	0.00	5.00	5.00
Total Composite Type	9.00	10.00	5.00	5.00
Average Composite Type Length	1.00	1.20	5.00	5.00
Number of operators	55.00	44.00	5.00	5.00
Number of predicates	29.00	21.00	5.00	5.00
Average operators perschema	18.33	22.00	5.00	5.00
Average operators perpredicate	1.90	2.10	5.00	5.00
Average predicates perschema	9.67	10.50	5.00	5.00
Number of Operation Schema	0.00	0.00	5.00	5.00
Number of Observation Schema	2.00	1.00	5.00	5.00
Average Schema Inclusion	0.67	0.50	5.00	5.00
Average Depth of Schema	0.67	0.50	5.00	5.00
Average variables perschema	16.00	16.00	5.00	5.00
Total Complexity			100.00%	

Figure 3-14 : Example of output from complexity marking

The above examples of output are presented using the first (numeric) type. An example of a result which is shown verbosely is given in Figure 3-15.

Complexity Marks
You have done excellently in Number of distinct identifier You have done excellently in Number of Schema   You have done excellently in Average Depth of Schema You have done excellently in Average variables perschema Score 100.00

Figure 3-15 : Example of output from complexity marking

Below is an example of the third type of output, in which only the overall mark is shown.

Score 75.00

Further features

The system is able to give a reason for marks lost, when tested against a particular test case. If the schema should be true when it fails, then the system can show which predicate caused the error. Furthermore, if the schema succeeds the test, but fails in some of the analysis, the system is able to inform of the analyses that fail.

3.5. Conclusion

We have described in this chapter a system that gives a quality grade to a specification that is written in the Z language. Four factors are involved in determining the quality of a specification : typographic arrangement of a specification, complexity of a specification, static correctness of a specification and dynamic correctness of a specification.

References

1. A.M. Zin, *ZFDSS: A Formal Development Support System based on the Liberal Approach*, 1994. PhD Thesis, University of Nottingham, UK
2. Z. Shukur, E. Burke, and E. Foxley, "Automatic Marking System for Z Specifications," *Proceedings of PROGRESS 98 Conference*, Nottingham, UK, March 1998.
3. Z. Shukur, E. Burke, and E. Foxley, "The Automatic Assessment of Formal Specification Coursework," *Journal of Computing in Higher Education*, Amherst, Massachusetts, US, (will be published in) August 1999.
4. N.P.H. Haigh, "Providing tool support for Z," *Software Tools: Improving Applications*, pp. 185-191, June 1987.
5. J. M. Spivey, *The Z Notation : A Reference Manual*, 1989. Prentice-Hall, Inc.
6. C. Jones, "A Survey of Programming Design and Specification Techniques," *Proceedings of IEEE Conference on Specification of Reliable Software*, pp. 91-103, Cambridge, Mass., 1979.
7. A.M. Gravell, "What is a Good Formal Specification?," *Fifth Annual Z User Meeting*, Oxford, UK, 17 December 1990.
8. R. Macdonald, "Z Usage and Abusage," *Report 91003 Royal Signals and Radar Establishment*, Malvern, Worcs, February 91.
9. B. Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems*, Winthrop Publishers, 1980.
10. R.S. Pressman, *Software Engineering*, McGraw-Hill Company Europe, 1992.
11. A. Diller, *Z An Introduction to Formal Methods*, John Wiley & Sons, 1994.
12. L. Bottaci and J. Jones, *Formal Specification Using Z: A Modelling Approach*, International Thomson Publishing, 1995.
13. "ZUM '98: The Z Formal Specification Notation," *Proceedings of 11th International Conference of Z Users*, no. 1493, Springer, September 1998.

14. M.J. Rees, "Automatic Assessment Aid for Pascal Programs," *SIGPLAN Notices*, vol. 17 , no. 10, pp. 33-42, October 1982.

15. M. Coleman and S. Pratt, *Software Engineering for Students 1986,* Chartwell-Bratt Ltd., 1986.

16. J.M. Spivey, *The Z notation : Reference Manual,* Prentice Hall, 1988.

Chapter 4
Inspecting the Correctness of Specification through System-state Analysis

4.1. Introduction

Testing techniques for assuring the correctness of a formal specification have been proposed by many people, for example by Kemmerer[1] and Jalote.[2] The fundamental requirement of any software testing is to define the objectives accurately. The testing objectives help one plan design test cases in such a way that the objectives can be met.[3] A number of rules that can serve as a basis for testing objectives are described by Myers[4] and are presented as below.

- Testing is a process for executing a program with the intent of finding an error.
- A good test case is one that has a high probability of finding an as yet undiscovered error.
- A successful test is one that uncovers an as yet undiscovered error.

Information flow for the testing process which is adapted from[3] is shown in Figure 4-1. From the figure, it can be that seen the process has two classes of inputs; software configuration and test configuration. A "test configuration" includes both test cases and expected results.

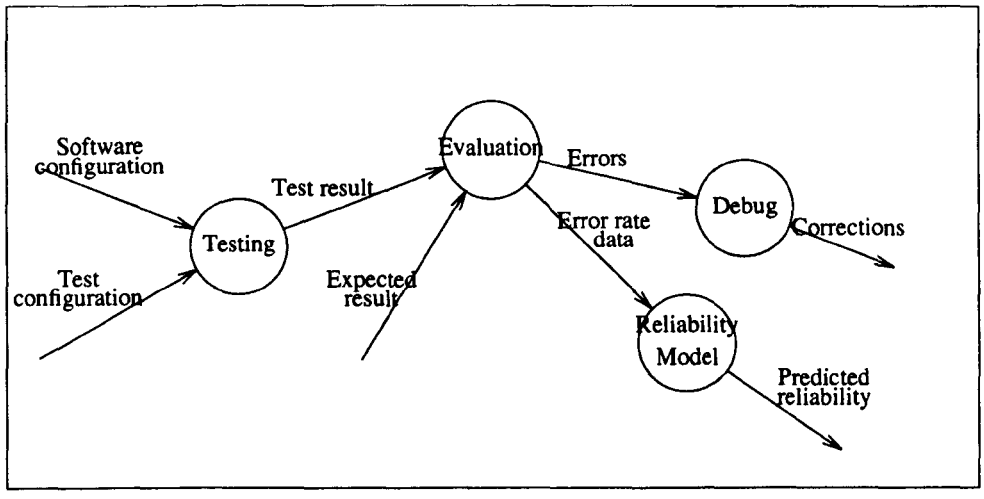


Figure 4-1 : Test Information Flow

Mandrioli[5] says that "many testing techniques address the problem of generating test data (for example program inputs) for a system but do not address the problem of verifying the correctness of the outputs obtained during the testing activity." Richardson[6] tackles this problem by incorporating *oracles* into the testing process. She says, "a test oracle determines whether a system behaves correctly for test execution."

Another issue is in debugging, which is defined as a process of removing an error.[3] According to Beizer[7] it, "occurs as a consequence of successful testing" , that is when a test case uncovers an error. It has been said by Shneiderman[8] that this process is one of the most difficult parts of software engineering. This process is normally supported by debugging tools such as debugging compilers and dynamic debugging aids (sometimes called "tracers"). In terms of program debugging methodologies, these are divided into two main kinds; *static analysis* that examines the code, and *dynamic analysis* that examines the running of the code on specific examples.[9] An outcome of the debugging process might be the knowledge of the location and the type of all bugs and the corrections necessary.[9]

This chapter discusses how the above idea can be used in testing a Z specification in a learning environment. With respect to the information flow in Figure 4-1, we 'execute' a specification against selected test cases and evaluate the results of the testing by comparing them with the expected results using a test oracle. The result of the evaluation is presented by using some numeric measurement such as a percentage figure. The value of the percentage can give an idea about the degree of the correctness of a specification. When an error is discovered, the debugging process can be initiated.

We can test the correctness of the system specification behaviour by analysing the state of the system because Z specifications use a state-based approach. The execution behaviour of a specification can be demonstrated by animating it. To make the specification animatable, we translate the specification into Prolog using Zin's Z to Prolog translator, `zp`.[10] We produce a percentage that shows the level of correctness of the specification. When the level of correctness is not at an acceptable level (we may not demand complete correctness), some faults are shown to exist in the specification. To 'debug' a specification, we developed a debugging tool to locate the error.

The work presented in this chapter will be submitted to the IEEE Transactions on Software Engineering.[11]

4.2. The system-state analysis approach

In principle, the specification analysis can be classified into two areas: pre-condition analysis and post-condition analysis. Furthermore, post-condition is analysed in two ways; abstract analysis and concrete analysis. These analyses can be performed by investigating only the system-state. The results of each analysis are presented as an integer. The greater the value of the integer the more successful the analysis.

4.2.1. Pre-condition analysis

Predicates which define the pre-condition of a schema can be obtained by hiding all the primed and output variables. By only using unprimed system variables (i.e system-state variable and input variables), we can detect the occurences of respective pre-conditions in a schema. This can be done by using a combination of test cases. The combination of results infers the condition possessed by the specification. Marks can be awarded according to the completeness of the pre-conditions that the specification possesses.

We illustrate this situation by using the schema *AddTelephone*. The schema consists of two pre-conditions: the input *name* must not be in the database, and the input *phone* must be valid data. By using a technique proposed by Hall[12] , we produce the table below (Table 4-1):

	TD1	TD2	TD3	TD4
known	-	-	-	-
telephone	-	-	-	-
name?	$\notin known$	$\in known$	$\notin known$	$\in known$
phone?	valid telephone	invalid telephone	invalid telephone	valid telephone
known'	-	-	-	-
telephone'	-	-	-	-

Table 4-1 : Test data classification

An example of a system-state that satisfies TD1 is:

$known = \{ali, abu\}$

$telephone = \{(ali, 9422144), (abu, 9788777)\}$

$name? = ahmed$

$phone? = 9514232$

An example of a system-state that satisfies TD2 is:

$known = \{ali, abu\}$

$telephone = \{(ali, 9422144), (abu, 9788777)\}$

$name? = ali$

$phone? = scotland$

An example of a system-state that satisfies TD3 is:

$known = \{ali, abu\}$

$telephone = \{(ali, 9422144), (abu, 9788777)\}$

$name? = ahmed$

$phone? = england$

An example of a system-state that satisfies TD4 is:

$known = \{ali, abu\}$

$telephone = \{(ali, 9422144), (abu, 9788777)\}$

$name? = abu$

$phone? = 9515455$

If we succeed in using test data 1 (TD1) for the specification, it does not guarantee that both pre-conditions are correct. It could be that either both pre-conditions exist, or neither exist, or only one exists. If the test data were to fail then either both of the pre-condition exist or one of the pre-condition exists. This situation is similar to program testing, where R.A. DeMillo et al[13] said "if the program gives the incorrect answer, then certainly the program is in error. On the other hand, if the program gives the correct answer, it may be that the test data is not sensitive enough to distinguish that error." However if we extend the testing by using test data TD2, TD3 and TD4, and all three produce negative results, this combination of results infers that the specification has both pre-conditions.

By generalising the *AddTelephone* problem into any problem that contains two pre-conditions A and B, we derive four different instances of test data as follows: test data that satisfies pre-condition A and B (tcAB), test data that does not satisfy pre-condition A but satisfies B (tc¯AB), test data that does not satisfy pre-condition B but only A (tcA¯B), and finally test data that does not satisfy either pre-condition A or B

(tc~A~B). From those four instances, we can write several combinations as in the table below (Table 4-2).

Comb.	tcAB	tc~AB	tcA~B	tc~A~B	inference	Marks	Rationale of marks given
1	/	x	x	x	A & B	100%	satisfies both pre-conditions
2	/	x	/	x	A	75%	satisfies only one pre-condition
3	/	/	x	x	B	75%	satisfies only one pre-condition
4	x	/	x	x	~A & B	50%	satisfies one pre-condition and does not satisfy the other one
5	x	x	/	x	A & ~B	50%	satisfies one pre-condition and does not satisfy the other one
6	x	x	x	/	~A & ~B	0%	does not satisfy both pre-conditions
7	x	/	x	/	~A	0%	does not satisfy one pre-condition
8	x	x	/	/	~B	0%	does not satisfy one pre-condition
9	/	/	/	/	-	0%	no pre-conditions exist

Legend :
/ : Success when trying against a specification
x : Fail when trying against a specification

Table 4-2 : Combinations of test data

If the schema consists of complete pre-conditions, it will satisfy the first combination (in Table 4-2), and should be awarded full marks. By a complete pre-condition, we mean the specification consists of predicates which cover the definition of pre-conditions in the model solution (which is provided by the teacher). The marks accompanying each combination are based on the teacher scheme presented in the last two columns of the table. For example, if a student testing corresponded to combination three, then 75% of the marks would be awarded.

However, this analysis technique has a high cost. If a schema has N pre-conditions, it will require 2^N different test cases, which means that the combination of results also increases exponentionally. Therefore it is not practical to produce a marking scheme by creating a table of combinations. To overcome this problem, we propose another scheme with the distribution of the marks as presented in Table 4-3.

Test Data	tcAB		tc~AB		tcA~B		tc~A~B	
Result	/	x	/	x	/	x	/	x
Marks	5	0	0	5	0	5	0	5

Table 4-3 : Marking scheme for first combination of test data

If the student specification succeeds when tested against test case tcAB, he/she will be awarded 5 marks, and if the specification failed there would be no marks awarded. However, if the specification failed when tested against test case tc~AB, tcA~B or tc~A~B, he/she would obtain 5 marks, and 0 otherwise for each of the tests. These marks would be totalled up to provide the overall mark for the pre-condition analysis.

The two marking schemes are similar in several aspects. In the first 5 combinations the schemes do not differ. The difference lies in the last four combinations. The scheme in Table 4-2 awards no marks for those combinations. However, for combination 6, the student can get 50% if using the scheme in Table 4-3 and 25% (5 out of 20) for the last three combinations. Even though the schemes are not that different, scheme 2 offers a practical solution to the problem whereas scheme 1 cannot be implemented for realistically sized cases.

We examine the post-condition as described below only if the specification passess the ideal pre-condition analysis (tcAB).

4.2.2. Post-condition analysis

Post-condition analysis involves checking the system state after certain operations. After we have 'executed' the specification against selected test data, we can then analyse the system variables. The system variables are inspected in order to see that the 'action' carried out by the schema is accurate. For example, if the schema is performing the addition of an item of data in a state variable system, we will inspect whether the data has been added correctly, by analysing the system state after the 'execution' of the schema. We can check this in two ways (known as 'abstract' and

'concrete'). Abstractly, we check the existence of a relation between the system-state and the input or output. Concretely, we check the exact value of the system-state itself. An example of abstract analysis is

$ali \in known'$

and for concrete analysis

$known' = \{abu, ali, ahmad\}$

For every successful analysis, we set a corresponding mark. For example

Give 5 marks if the relation $ali \in known'$ is true.

Give another 5 marks if the relation $known' = \{abu, ali, ahmad\}$ is true.

4.3. The system approach

The definition of the system state shown by the schema is represented by predicate formulae. To test that the schema is defining the behaviour of the system correctly, we invoke the schema with a chosen set of values of the *system variables* (these may include global pre- and post- variables, as well as input and output variables). These variables (set with an instantiated value) are what we call *system inputs.* A *test case* is a reference to system inputs. Even though a test case can consist of output variables and post-variables, in principle it is better to prepare the test case just for input variables and pre-variables. We will use the term *input variable* for the literal meaning of input to a schema and *output variable* for the literal meaning of output from a schema.

If the schema is succesfully invoked (which can be classified as the pre-condition stage of analysis), we continue by analysing the values of the rest of the system variables (which can be classified as the post-condition stage of analysis). This principally involves *system outputs* (output variables and post-variables). Indirectly, this analysis corresponds to the concept of an oracle. Marks awarded to the success analysis are determined. This analysis information together with the marks are represented by a term which we choose to call *test weight.* A measure of the system's

behaviour is determined by the expected values of the system outputs. The complete analysis of the schema will involve a number of tests; some marks will be awarded for each test, and the total mark represents (in a certain sense) the overall success of the schema.

In order to reduce the complexity of the system test, Sadeghipour[14] proposed that the analysis should be better carried out on a schema one at a time. We use Hall's[12] idea for generating test cases and test weights. To select test cases, firstly we define the test domains and then for every test domain, "typical" elements (the value of input variables and pre-variables) are selected. This is done manually. To determine test weights, we need abstract analysis, concrete analysis and corresponding marks. Concrete analysis can be carried out automatically by running the model answer against selected test cases. Abstract analysis is carried out manually, and marks are set as described in the previous section.

4.4. Z specification testing system

Our system consists of four main testing processes: generating the test case, generating the test weight, testing and evaluation, and debugging.

Generating a test case

The process of generating a test case is not fully automated. Figure 4-2 shows the process flow. A model specification will be used to derive system input variables. This can be represented as

$$
\begin{aligned}
&\mathit{SchemaName} \\
&\mathit{variable}_1 = \_A \\
&\mathit{variable}_2 = \_A \\
&\quad \cdot \ \cdot \ \cdot
\end{aligned}
$$

where

SchemaName is the name of the schema that we want to test,

variable $_1$ etc will be the name of an input, output or pre- or post- state variables and,

_A is a temporary value for the variables. The real value for the variables will be input manually (by the teacher).

The test case then will be compiled to ensure the syntax (the format of the file) is correct. If so, it will be translated into Prolog to become a *compiled test case.*

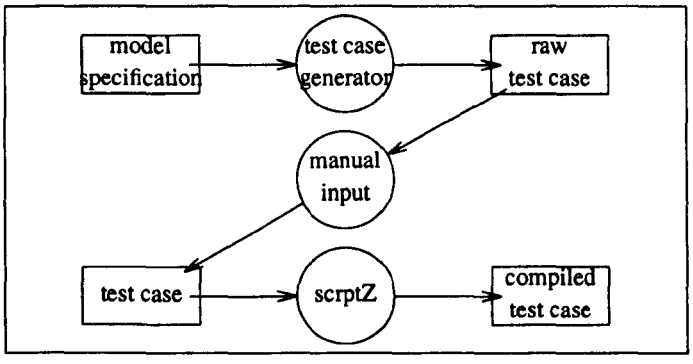


Figure 4-2 : Process of generating test case

Generating a test weight

The process of generating the concrete analysis is done by running a model specification against a test case. The format of the initial test weight file will be

SchemaName

0 : *concrete predicate* $_1$

0 : *concrete predicate* $_2$

. . .

The mark in this file is set automatically to zero. The teacher will set the abstract analysis together with the marks manually. The final test weight file will be

SchemaName

$mark_1$: *abstract predicate* $_1$

$mark_2$: *concerete predicate* $_1$

$mark_3$: *abstract predicate* $_2$

$mark_4$: *concrete predicate* $_2$

. . .

where

SchemaName is name of the schema that we have tested and

$mark_1$ is the mark we want to give if *predicate* $_1$ is *true.*

This complete test weight will be compiled to check the syntax and then will be translated into Prolog. Figure 4-3 shows the overall flow of the process.

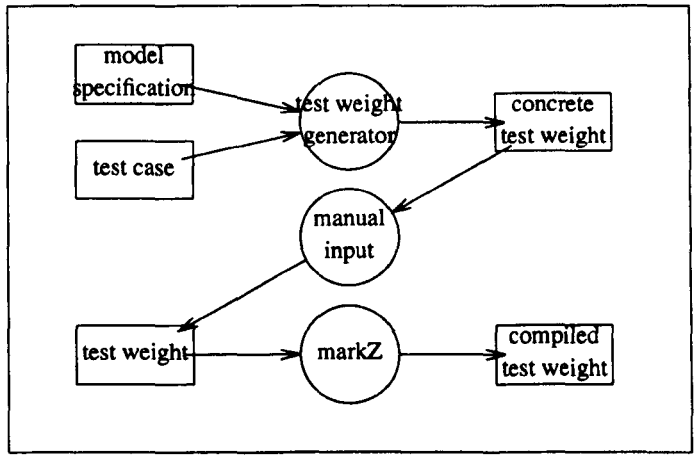


Figure 4-3 : Process of generating test weight

Preparation of the specification

Before any specification can be tested, the system will make sure that it has all the properties that are needed. The following are checked before testing is carried out;

schema name,

input and output variables, and

interested state variables

A message will be produced to say that the specification is ready to be used in the testing process or otherwise.

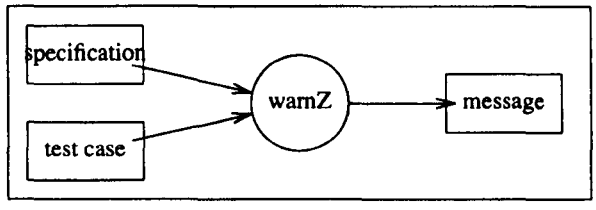


Figure 4-4 : Process of preparation

Testing and evaluation

Figure 4-5 shows the testing and evaluation process. The output of this process is a number that represents the level of correctness. If the specification does not obtain full marks then the debugging process can be carried out.

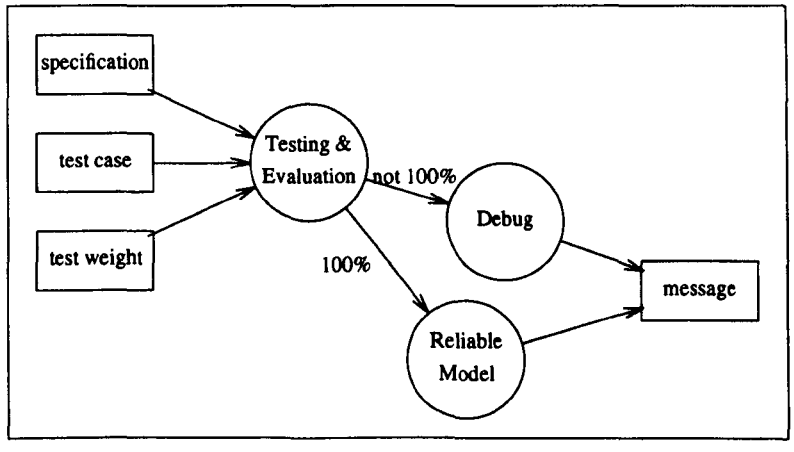


Figure 4-5 : Process of testing and evaluation

Debugging

The debugging process attempts to detect the reasons for lost marks. Marks may be lost during (post-condition) analysis or during 'execution' or for a number of other reasons. Marks are lost during analysis when one or more of the concrete or abstract post-condition analyses are not satisfied. If this is the case then the debugger will produce a message showing the one that is not satisfied. Marks are lost during execution when a particular test case fails when it should have succeeded. This normally happens because of the existence of an invalid operation in the specification. Other reasons for lost marks are:

- a particular test case, succeeded when it should not have;
- a run time error with the Prolog system.

Figure 4-6 shows the information flow of the process.

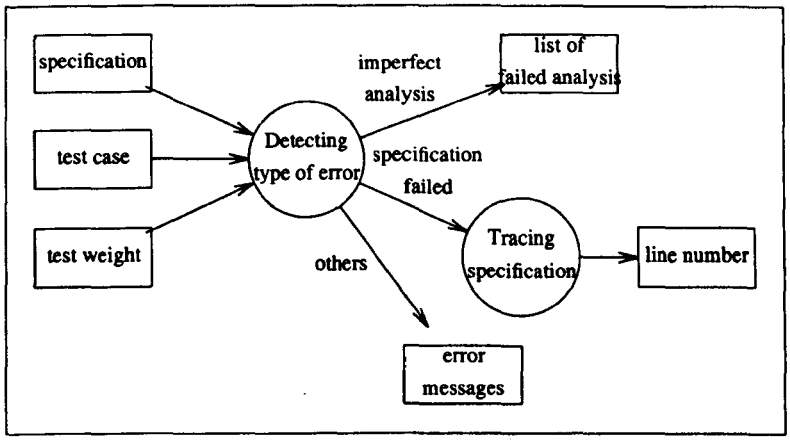


Figure 4-6 : Process of debugging

Supporting program

This section will explain the program that supports the above process.

Test case generator : `tdg`

The system assists the teacher in preparing the test case by providing the files which have all the system inputs (i.e system-state and input variables). Program `tdg` does this task. It takes a model specification and schema name as input and produces a file (with extension .tc) with the information of all the system inputs that should be associated with a value.

The syntax for `tdg` is :

```
tdg file1 testcase schema
```

where:

file1 is model specification, with an extension .z,
testcase is a name that will be used as a test case file name, and
schema is a schema to be tested.

Test weight generator : `tog`

Test weight consists of abstract analysis and concrete analysis of system variables as well as distribution of marks for each analysis. The system will prepare the concrete analysis for the teacher by using program `tog`. It takes a model specification and test case file as input and produces a file (with extension .w) with the information for the concrete analysis that will be carried out and the mark that will be awarded if that analysis succeeds.

The syntax for `tog` is :

```
tog filename testcase schema
```

where:

file1 is a model specification (with extension .z),

testcase is a file (with extension .tc) that will be used, and
schema is a schema to be tested.

Test case to Prolog translator : `scrptZ`

Before a test case (file with extension .tc) can be used, it needs to be translated to Prolog code by program `scrptZ`. During the translation process, program `scrptZ` also checks the syntax and semantics of the testcase (refer to *Expression* in Appendix A). A typical example of a syntax error is missing parentheses. An example of a semantic error is when the number of output parameters for respective schema is not the same as for the model. If the test case is succesfully compiled, it will produce a formatted file (with extension _tc.pl).

The syntax for `scrptZ` is :

```
scrptZ file1 file2
```

where:

file1 is model specification (with extension .z), and
file2 is test case script (with extension .tc).

Test weight to Prolog translator : `markZ`

Before a test weight can be used, it needs to be translated into Prolog code, by program `markZ`. During the process, `markZ` will check the syntax and semantics of the test weight (Appendix A). If the test weight is succesfully translated, it will produce a formatted file which is named with extension _w.pl.

The syntax for `markZ` is :

```
markZ file1 file2
```

where:

file1 is test weights (with extension .z), and
file2 is model specification (with extension .w).

For every test case and test weight file, we can only have one schema to test.

NOTE: Prolog recognises the values as small letters, therefore the parameters in the testcase should be written in small letters.

Specification and test cases compatibility checker : `warnZ`

The system also provides a facility to check the compatibility between student schema and the testcase. Program `warnZ` checks whether the schema name, number of inputs and outputs and the rest of the parameters in the student specification is the same as the model specification and the test case.

The syntax for `warnZ` is :

```
warnZ file1 file2 file3
```

where:

file1 is student specification (with extension .z),
file2 is test case (with extension .tc), and
file3 is model specification (with extension .z).

Debugger program : `dynpred`

The algorithm for the debugging process carries out the following checks:

- It checks whether the specification is prepared for the testing (using *warnZ*). If it is not prepared, an error message such as the following will be produced.

```
Warning error occured during testing
```

- It checks whether the schema can be animated against the given test data. If it failed when it should have succeeded, the error message

  ```
  Your animated specification has failed when tested
  against respective schema
  ```

  will be produced. Or if it is succeeds when it should have failed,

  ```
  Your animated specification should fail when tested
  against respective schema
  ```

  will occur. If it is caused by the former, the *tracing program* will be invoked and will determine the predicate which started the error. A message such as the one below will be shown.

  ```
  Error occurs at line number N
  ```

- It checks every analysis in a given test weight file, listing the analyses that failed. A message such as the following will be shown.

  ```
  Your animated specification did not satisfy the analysis i
  ```

The syntax for `dynpred` is :

```
dynpred file1 file2 dir [b | f | s]
        [-p | -pd | -c | -m | -md]
```

where:

file1 is a specification (with extension .z),
file2 is a test case (with no extension),
dir is a directory where the test case resides,
b | f | s : type of tracing speed, and
-p | -pd | -c | -m | -md : type of output.

The tracing program traces which predicate produced the error. Generally in Prolog, when a predicate fails the proceeding predicates will not be executed. The tracing program is used to trace the predicate in Prolog that caused the error and therefore refers it to the predicate in the Z schema (which was translated into that predicate). This is best illustrated as follows:

Predicates in Z

```
1:  DOM video_titles = videos
2:  videos_in UNION DOM video_out_to = videos
```

are represented in the Prolog implementation as

```
1:  dom(V322,V707),
2:  equalset(V707,V321,true),
3:  dom(V324,V709),
4:  union(V323,V709,V710),
5:  equalset(V710,V321,true),
```

The predicate at line 1 in Z is represented by the predicate at lines 1 and 2 in the Prolog implementation. The predicate at line 2 in Z is represented by the predicate at lines 3, 4 and 5 in the Prolog implementation. If the predicate at line 4 in the Prolog implementation fails, then the predicate at line 2 in the Z implementation also fails.

There are two steps to carry out this process; first to detect the predicate in the Prolog implementation that started the failure, and second, to interpret the result to represent the predicate in Z. To carry out the first step, we are dealing with a tracing issue. The Prolog system that we used (i.e *sicstus*) has a tracing facility. If it is done automatically, it will produce a robust tracing output, i.e every single predicate will be shown. An example is presented below:

```
Call: pVideo_shop ?
  Call: update(vcustomers,_655) ?
    Call: update(vcustomers) ?
      Call: name(vcustomers,_1097) ?
```

```
Exit: name(vcustomers,[118,99,117,115,116,...]) ?
Call: conc([118,99,117,115,116,...],[80],_1091) ?
```

If we want to have a first level tracing, we have to control it by using the *skip* command. An example is:

```
Call: pVideo_shop ?
Call: update(vcustomers,_655) ? s
Exit: update(vcustomers,[]) ?
Call: update(vcust_name,_649) ? s
Exit: update(vcust_name,[]) ?
Call: update(vcust_address,_643) ? s
Exit: update(vcust_address,[]) ?
```

In our case, we are interested in the first level of tracing. Therefore we developed a metaprogram (written in Prolog and called *metafail*) which traces the Prolog execution and produces an output consisting of the first level result automatically. The second step is to translate the representation in terms of a line in the Z schema. This is easily done. The program that handles this is called `obtraceZ`. Both of the programs *metafail* and *obtraceZ* are called internally by the program *dynpred.*

Testing and evaluation program: `dyncorr_z`

The main program, `dyncorr_z` is written in the C programming language. The program uses several files, both from the teacher and from the student. The program also uses the Prolog runtime library and calls on previously discussed programs.

Some actions need to be carried out before the process can be invoked. The model solution must be named as *model.z*. The student specification and model solution should be translated to a Prolog implementation (by using `zp`). It is advisable to use the program `warnz` to ensure that both of the specifications are ready to be used in the process. All the files provided by the teacher (such as the model solution, test weight and test data) must be located in the same directory. All the respective files

should be in the Prolog implementation. This can be done by using previously described programs. As regards the calling of the Prolog predicate from the C code, the respective Prolog program must have the Prolog object files. This can be done by compiling the Prolog program using a built-in predicate in the Prolog system, i.e *fcompile*.

The syntax of the command for `dyncorr_z` is :

```
dyncorr_z -d dir1 [-v0 | -v1 | -v2 | -v3 ] [-f] file1
          -tN test1 test2 ..testN
```

where:

dir1 is the directory where model.z, weights file (file with extension .w), test data (file with extension .tc), zdef.pl and model.dv are all located,
file1 is Z specification documents (with extension .z),
-v0 | -v1 | -v2 | -v3 : result shown verbosely,
-f : result shown in sentences,
-tN : testcase flag, showing that the next N arguments is a test case files, and
-d : directory flag, showing that the next argument is a directory.

4.5. Conclusion

In this chapter, we have shown how we can apply system-state analysis to determine the correctness of a specification. We extended the research by presenting an idea on how to scale the correctness in terms of a percentage figure. This technique of testing a specification has been one of the important components in the automatic marking system for a Z specification (see Chapter 3) written by a student. Having this technique is important as it can support an on-line learning environment.

References

1. R.A. Kemmerer, "Testing Formal Specifications to Detect Design Errors," *IEEE Transactions on Software Engineering*, vol. 11, no. 1, pp. 32-43, January 1985.
2. P. Jalote, "Testing the Completeness of Specification," *IEEE Transactions on Software Engineering*, vol. 15, no. 5, pp. 526-531, May 1989.
3. R.S. Pressman, *Software Engineering*, McGraw-Hill Company Europe, 1992.
4. G. Myers, *The Art of Software Testing*, Wiley.
5. D. Mandrioli, S. Morasca, and A. Morzenti, "Generating Test Cases for Real-Time Systems from Logic Specifications," *ACM Transactions on Computer Systems*, vol. 13, no. 4, pp. 365-398, Plotecnico di Milano, November 1995.
6. D.J. Richardson, S. Leif Aha, and T.O. O'Malley, "Specification-based Test Oracles for Reactive Systems," *Proc. of the 14th ICSE International Conference on Software Engineering* , pp. 105-118, IEEE/ACM, New York, 1992.
7. B. Beizer, *Software system testing and quality assurance*, Van Nostrand Reihold, 1984.
8. B. Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems*, Winthrop Publishers, 1980.
9. C.K. Looi, "Automatic program analysis in a Prolog intelligent teaching system," *PhD Thesis*, University of Edinburgh, May 1988.
10. A.M. Zin, *ZFDSS: A Formal Development Support System based on the Liberal Approach*, 1994. PhD Thesis, University of Nottingham, UK
11. Z. Shukur, E. Burke, and E. Foxley, "Inspecting the correctness of specification through system-state analysis," *IEEE transaction on Software Engineering*, (will be submitted) 1999.
12. P.A.V. Hall, "Towards Testing with Respect to Formal Specifications," *Proc. of Second IEE/BCS Conference: Software Engineering 88*, pp. 159-163,

London, 1988.

13. R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on test data selection: Help for practiting programmer," *Computer*, April 1978.

14. S. Sadeghipour, "Test Case Generation on the basis of Formal Specifications," *Proc. FEmSys '97*, 1997.

Chapter 5
Managing Z Specification Coursework On-line

5.1. Introduction

It has been said, "marking on paper is inappropriate for most computer-based learning. Administration of assessment is a time-consuming process, made more difficult by having to deal with large quantities of paper."[1] In general, "scripts can easily get lost, detached or damaged."[1] The use of automatic assessment tools[2–5] has already benefited the computer-based learning process. However, it should be noted that there are several other issues that need to be considered when implementing on-line specification coursework. It is beneficial to the student and teacher to have a system that can check the type and syntax of a specification, mark the exercises automatically, allow submission of the solution in electronic form and provide other facilities to support learning.

There are many studies that investigate improving educational systems, especially by incorporating aspects involved in teaching. One example is presented in a paper written by Woolf.[6] Here the author says that an Intelligent Tutoring System (ITS) will "allow the emulation of a human teacher in the sense that an ITS can know what to teach (domain content), how to teach it (instructional strategies), and learn certain relevant teaching information about the student being taught." The idea of the ITS architecture has been used in developing systems for use in military and aerospace research; for example Intelligent Computer-Assisted Training Testbed Program (ICATT)[7] , and Intelligent Computer-Aided Training (ICAT).[7] The idea has also been employed in educational institutions; for example ISIS-Tutor[8] (an adaptive hypertext learning environment) , Sherlock[9] (a coached practice environment for an electronics troubleshooting job) and ELM-ART[10] (an adaptive, knowledge-based tutoring system on the WWW that supports learning programming in LISP).

Ceilidh [11] is another example of a well-known educational system which is being used in many higher education institutions. The research upon which Ceilidh is based started at the University of Nottingham in 1988. Its essence is an automatic assessment mechanism for programming. The first course implemented in Ceilidh was a C programming course. At the moment there are various courses such as C++, Software Tools, Prolog and many others.[12]

The Z formal specification language has been taught at the university for a number of years. In previous years, the coursework was handled in the conventional way, that is using paper and submitting to the teacher for marking. This year the teaching system has been improved by implementing the coursework in Ceilidh. The coursework includes writing Z specifications, description of solutions in essay form, short-answer form and mathematical scripts. The Z specification can be either non-animatable, in which case we can only do static marking or they can be animatable in which case we can also do dynamic testing.

Z specification coursework is different from other courseworks in Ceilidh in several aspects. Specific components are needed to support the process of making the coursework on-line:

- Z specification employs mathematical symbols and other specialised symbols. An editor that can support those symbols in Z and embed the concept of WYSIWYG is therefore needed.
- A program is needed to ensure that the specification is free from syntactic errors. Unlike programming languages that have established compilers such as `gcc` for the C language, most Z type and syntax checkers were built for in-house use.
- In order to validate a specification, several techniques can be used, such as Formal Technical Review [13] , View Point Resolution[14] , symbolic execution[15] and testing.[16, 17] We require a method that can be automated, and have chosen to use testing techniques which animate the specification.

- Marking tools for assessing a specification automatically will remove problems encountered when using conventional hand marking, such as inconsistency in awarding marks. In addition, marks can be made available to students very quickly compared to the use of hand marking.

Incorporating the above tools into the existing course management system simplifies the managerial issues mentioned earlier.

This chapter will start with a short description of Ceilidh. For further details the reader can refer to Learning Technology Research group papers.[18] This will be followed by a description of the special tools used and then an explanation of the facilities offered by Ceilidh in handling Z specification exercises. The description will emphasise exercise facilities both for the student and the developer. The tools that are being used have been built by Nottingham University researchers.

The work has been presented at the Association for Learning Technology Conference '98 and the abstract was published in the proceedings.[19] It has also been informally discussed in the Educational Session of Z User Meeting '98. A full paper of this chapter has been submitted to the Journal of Computers in Mathematics and Science Teaching.[20]

5.2. Ceilidh : Course Management System

The core of the Ceilidh system is an automatic coursework submission and marking component which can give instantaneous feedback on student programs from perspectives such as program complexity and typographic style. Beyond this, Ceilidh also provides on-line access to all notes, examples, exercises and solutions as well as providing progress monitoring for tutors and general course administration for teachers.

Ceilidh helps the teacher in administrating courses by setting exercises, collecting students' work (in electronic form), detecting plagiarism, manipulating students'

marks, performing statistical analysis of marks and many other facilities. These facilities help the staff by reducing administration and marking time, therefore allowing greater effort to be focused on effective teaching.

The major differences among the courses in Ceilidh is at the exercise level. Besides the notes and courseworks, a complete new set of marking tools (and any necessary support tools) will need to be written when implementing a new language in Ceilidh. Different languages require different facilities in the exercise menu, and Ceilidh chooses the correct menu by looking at the file extension.

5.3. Components

Z specification editor

Since the Z language includes many special symbols that are not available on an ordinary keyboard, all Z schemas for this course are created in an ordinary ASCII file with a normal ASCII text editor such as `emacs` or `vi`. The format uses a technique related to the UNIX roff text processing system, and forming an extension to the existing mathematical pre-processor.[21] The specification can be viewed in ordinary Z format by using the Z pre-processor `zpp`.[22] This is a roff pre-processor for printing Z specifications. `zpp` takes a Z specification written in Z-roff format and transforms it into PostScript form which is viewed using `ghostview`.

Type and syntax checker

A type and syntax checker, `zc` that has been developed by Zin[23] is used. `zc` takes a Z specification written in Z-roff format and reports errors. The compiler classifies the errors into three types; lexical errors, syntax errors and semantic errors. The compiler will display the message

```
The specification has been accepted
```

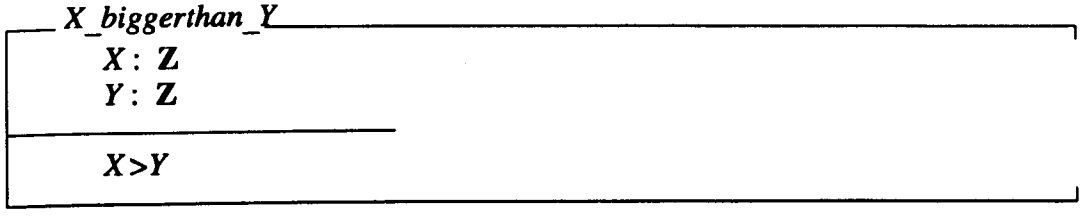
if there is no error in the specification. Otherwise a list of information about the errors will be displayed. This includes the line number where the error was detected followed by a description of the possible cause of the error, and the total number of errors at the last line. For example:

```
35 : In function INTER type not compatible
63 : name used before declaration - "ok"
There are 2 errors in the specification
```

Animator

The specification is translated into Prolog code by using the Z-Prolog translator zp[23] also written by Zin. This is used so that the student can validate their specification and increase their understanding of it.

For example, zp translates the schema



into the Prolog statement,

```
pX_biggerthan_Y :-
        greater(V301,V302,true).
```

Z marking tools

It is not necessary to have an automatic marker to handle the Z coursework on-line. However, if an automatic marker is incorporated in the system, it will reduce the problems associated with conventional marking. We have developed an automatic marking system to assess Z coursework automatically. In general, the system will

receive a Z specification written in Z-roff format with some particulars from the teacher, and then produce a mark. It is a combination of several tools which have specific tasks.

- Testing the *dynamic correctness* of a specification. The specification is 'executed' in order to check whether it performs the required task. This is done by animating the specification against sets of test cases and marks are awarded based on the correctness of the animation. Queries are provided by the teacher to check on the correct relationship between input and output variables.
- Marking the *static correctness* of a specification. The score for our static analysis will be based on the outcome of applying `zc` and the occurrences of other static problems in the specification.
- Assessing the *layout* of a specification. Many researchers have discussed the layout of Z specifications.[24–27] The system incorporates a selection of the proposed features.
- Marking the *complexity* of a specification. Considering that there are many ways to describe the same problem, the complexity factor should be taken into account when assessing a specification. Each complexity metric is marked relative to the corresponding counts for the model specification.

These tools do not merely produce marks. The tools are also able to inform the student where marks have been lost and why. The system can advise the student on which predicate caused the error during dynamic marking. For more detailed information, refer to chapter 3 and chapter 4.

5.4. Z specification coursework on-line

The tools discussed in the previous section have been incorporated into Ceilidh. The overall functionality of Ceilidh is categorized into student facilities and course management facilities. Students and teachers use the system to obtain access to course resources such as notes, work and lecture schedules. Course management

involves the process of developing the course, setting up the course and exercises and monitoring the running of the course. In the Ceilidh system, these three facilities reflect three categories of staff involved in managing a course: course tutors represent teaching assistants and are provided with additional facilities to inspect work and to summarize the progress of individuals or groups of students ; course teachers are provided with facilities to administer entire courses; and course developers can amend the course information including notes, exercises, and marking metrics.

The following sections are a discussion about student and course developer facilities at the exercise level.

5.4.1. Student facilities in handling an exercise

In the Z course, Ceilidh provides the following key facilities to students:

[i] It allows students to view general temporal course information such as hand-in times for course work and more permanent information such as lecture notes. This information can be viewed on-line or printed.

[ii] It provides access to a number of exercises in each unit. Each exercise contains a question and other information. The question forms a reasonably precise definition of the problem which the student has to solve. A typical question might be the following.

> In the context of the "Student Class" schema, described in the notes, write a schema for the operation Add_Student defined as follows:
> There shall be a command to add a new student to the class. The name of the student will be an input to the command. It is to be assumed that the new student has not handed in any work. The new student must not already be a member of the class.

[iii] It offers an outline (skeleton) Z specification source. The skeleton may be as helpful as the teacher wishes; it may contain very little information, or may be given in such a way that students need to fill in only the predicate part of the schema. A skeleton is necessary if the automatic marking system is to be used because AZAS will search for specific properties in the answer such as a schema name.

[iv] Ceilidh allows students to edit the ASCII form of the specification, and to view and print it in graphic form.

[v] Once the schema file has been created, the student can type check and syntax check it. If (and only if) these checks are successful, the schema can then be translated into Prolog, so that it can be animated for further testing.

[vi] The student can test the correctness of their specification dynamically against the exercise test data. This is not the same as "run" in programming languages where the interest of the student is to see the output of the test-run for the given input. The result of the test in our case is only a comment such as whether their specification is successfully tested.

[vii] When the schema has been completed and tested to the student's satisfaction, it can then be submitted to the system. The system will retain a copy of the submitted work, and may, where appropriate, mark it and return marks to the student to help them to assess their specification quality. Before any marking action is invoked, the system will first repeat the type and syntax checking, followed by translation to Prolog code. Here, the Z automatic marker plays its role, which is to mark the specification using a number of different quality factors determined by the teacher.

[viii] Ceilidh may allow the student to view a model solution, to view test data and to animate their solution against the test data. The model solution can be seen only after the deadline for submission has passed.

[ix] Students can view their work through a PostScript viewer; our viewer is called `ghostview` and includes a printing option.

[x] It allows them to comment on a specific exercise or on the system as a whole. Comments within the course are sent by email to the course teacher. In particular, whenever students submit work, they are asked (if they want) to comment on the marks obtained.

[xi] Ceilidh offers help facilities, including an overview of the marking metrics employed by the system and a good specification style guide.

Solving an exercise

The initial (system level) menu appears as shown in Figure 5-1. When the student selects a particular course, the menu shown in Figure 5-2 is displayed.

```
Ceilidh system top level menu:
lc    list course titles                 | sc    move to named course (pr1)
vcn   view course notes                  |
vp    view papers                        | pp    print papers (developer)
clp   change printer                     | h     for more help
co    make a comment to teacher          | q     quit this session
fs    find student                       | ft    find tutees
-----------------------------------------
System level command:
```

Figure 5-1 : System Level Ceilidh Menu : Z Course

```
Course and unit menu for course Z unit 4: Type
vp    view papers                        | pp    print papers (developer)
lu    list unit titles                   | su    set unit code
lx    list unit exercise titles          | sx    move to named exercise (1)
lux   list units and exercises           | state current exercise state
vn    view notes on the screen           | pn    print notes on brother14
csum  view course summary                | usum  view unit summary
vm    view all marks                     | co    make a comment to teacher
clp   change printer                     | motd  view motd
h     help                               | H     view student guide
kh    keyword help                       |
q     quit saving current status         | q!    quit Ceilidh, no saving
-----------------------------------------
Course level command:
```

Figure 5-2 : Unit and Course Level Ceilidh Menu : Z Course

At the course/unit level, the student is given access to look at the details of the course such as units, exercises and notes. The student can then move to the appropriate unit

and select a particular exercise. The menu shown in Figure 5-3 would then be displayed.

```
Z language menu for course Z unit 4 exercise glv: Type
vq    view question on the screen          | pq    print question on brother14
cq    copy question to file                |
co    make a comment to teacher            | set   set up coursework
h     for context help                     | H     for general help
q     to return to calling menu            | q!    quit Ceilidh
ed    edit your program                    | tc    type and syntax check
tp    translate Z to prolog                |
view  view post script                     | fresh   to refresh post script
rut   run yours against test data          |
sub   submit/mark your program             |
std   look at the test data                |
vs    view solution program                | ps    print (teacher) on brother14
cp    copy solution (teacher)              |
state check details of exercise            |
-------------------------------------------
Type compiled language command (default vq):
```

Figure 5-3 : Z Exercise Level Ceilidh Menu : Z Course

This is the level at which most of the student work will be undertaken. Each exercise will have been set up by the teacher, and will include a question, a skeleton solution, and all necessary testing information. A typical sequence of activities at this level might be as follows.

First, use `vq` to look at the question. The question can only be viewed in ASCII form. The student may need to study the question for a while before tackling it on the computer.

The student will then use `set` to set up a skeleton solution. This command typically puts an outline of the required specification into the student directory, to give them a flying start in solving the problem. In more complex exercises later in the course, it may set up other data files as well.

At this stage, the student can start to develop his/her solution, using the command `ed` to edit the specification in Z-roff format as described earlier. `ed` calls their preferred ASCII editor such as `emacs` or `vi`. This can be done by setting the environment variable **EDITOR** in their own login script file.

Next the student will probably view the PostScript version of the specification.

The student would then use `tc` to type and syntax check the specification. Even though this is not essential, later stages of the marking process will not function unless the specification passes these checks.

The next step is to use `tp` to translate the specification into Prolog. This action is not necessary unless the marking process involves dynamic correctness. It is essential if the student wants to test their solution against the exercise test data.

Once the student has successfully checked his/her specification, the system is ready to submit and mark. This is done using `sub`.

At any point the student can convert their solution to PostScript format and view it by using the `view` command. When the command is invoked, the student answer which is in Z-roff will be translated to PostScript format, and `ghostview` (the PostScript viewer) will be loaded and automatically displays the PostScript form of the student answer.

When the student updates his/her specification (which is in Z-roff format), the command `fresh` can be used to translate his/her updated specification to PostScript form. Assuming that the `ghostview` has been loaded using the `fresh` command, the student can use the facility `<Page><Redisplay>` in `ghostview` to view their updated specification in PostScript form. Figure 5-4 shows a student editing an early Z exercise and viewing a PostScript form of it.

Submission

When the student uses the `sub` command to submit and mark his/her solution, the computer's response will be something like that shown in Figure 5-5.

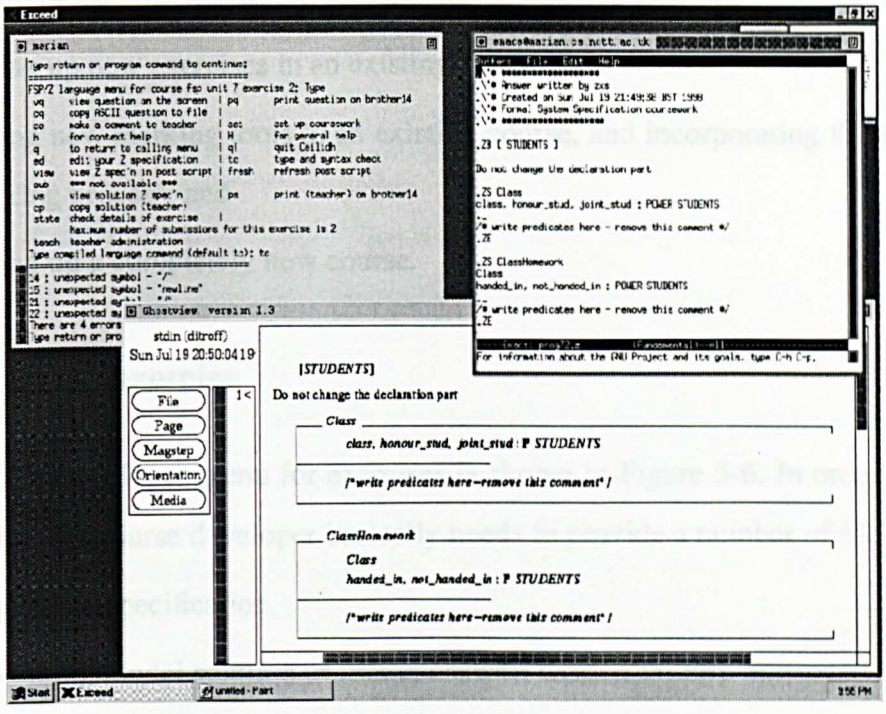


Figure 5-4: Student editing and viewing a Z exercise

Mark summary		
category:	mark:	out of
Typographics:	20:	20
Dynamic Correctness:	20:	20
Static Correctness :	20:	20
Overall mark awarded	100 out of	100

Figure 5-5 : System Output

5.4.2. Course developer facilities

The course developer has the most access to functionality in Ceilidh compared to other Ceilidh users.[11] Besides the above facilities, Ceilidh provides the course developer with additional facilities which include:

[i] setting up new units and editing notes. The notes are written in `zpp` format, and can be translated to PostScript.

[ii] changing details such as the test data for an existing exercise;

[iii] setting up new exercises in an existing course;

[iv] adding new marking tools to an existing course, and incorporating them into the marking process; and

[v] setting up a completely new course.

Setting up the exercise

The Ceilidh Developer menu for exercises is shown in Figure 5-6. In order to set up an exercise, the course developer basically needs to provide a number of files:

A question/specification.

A working model solution or model answer, written in `zpp` format.

A test cases file, to set up a data state for testing the schema. It is written in the format:

$$
\begin{aligned}
&\mathit{SchemaName}\\
&\mathit{variable}_1 \;=\; \mathit{value}_1\\
&\mathit{variable}_2 \;=\; \mathit{value}_2\\
&\quad\vdots
\end{aligned}
$$

where

SchemaName is name of the schema that we want to test,

$\mathit{variable}_1$ etc will be the name of an input, output or pre- or post- state variables

value_1 etc can be any value which that variable has to take

A file defining the weight to give to each test specifies the predicates we wish to test, and the mark to be awarded for each successful test. The format of a test weight file is as below:

$$
\begin{aligned}
&\mathit{SchemaName}\\
&\mathit{mark}_1 \;:\; \mathit{predicate}_1
\end{aligned}
$$

$mark_2$: $predicate_2$

. . .

where

SchemaName is name of the schema that we have tested

$mark_1$ is the mark we want to give if $predicate_1$ is *true*

The developer's exercise menu has three additional facilities compared with the programming version of the Ceilidh developer exercise menu.

```
Z developer's exercise menu (Course fsp, unit 6, ex evs):
et  to edit title                    | ety to edit type file
ep  to edit Z model                  | eq  to edit question
es  to edit skeleton                 | ef  edit program and fork
esa to edit setup actions file       | ema to edit mark actions file
tc  to type and syntax check         |
eth to edit tutor help file          |
tp to translate to prolog            | am  to animate
ss  to set static correctness        | sf  to set features mark
st  to set typographic weights       | sc  to set complexity weights
sd  to set dynamic mark scheme       |
mk  to mark the model soln           | mv  to mark more verbosely
mks to mark a studs soln             | mka to remark all studs
ae  to add essay actions             | rm  to remove essay actions
ch  to do a complete check           |
h   help                             | q   to return to outer level
-------------------------------------
Type developer's Z command: sd
```

Figure 5-6 : Developer's Exercise Menu : Z Course

The additional facilities are the type and syntax checking facility, the facility to translate a specification to a Prolog implementation and a window to carry out animation in a Prolog environment. This last facility is used to help the teacher to check that the model solution works properly when animated against selected test data. The window is shown as in Figure 5-7.

```
Animate
SICStus 3 #5: Wed Jun 11 17:19:31 BST 1997
{compiling /cs/u24h/ceilidh/ceilidh/course.fsp/unit.6/ex.evs/zdef.pl...}
{/cs/u24h/ceilidh/ceilidh/course.fsp/unit.6/ex.evs/zdef.pl compiled, 3420 msec 123520 bytes}
| ?-
```

Figure 5-7 : Window for Animation

5.5. World Wide Web version

For the purpose of presenting the system to remote users, we have developed a World Wide Web interface which includes the basic functions of solving a Z exercise. This is shown in Figure 5-8.

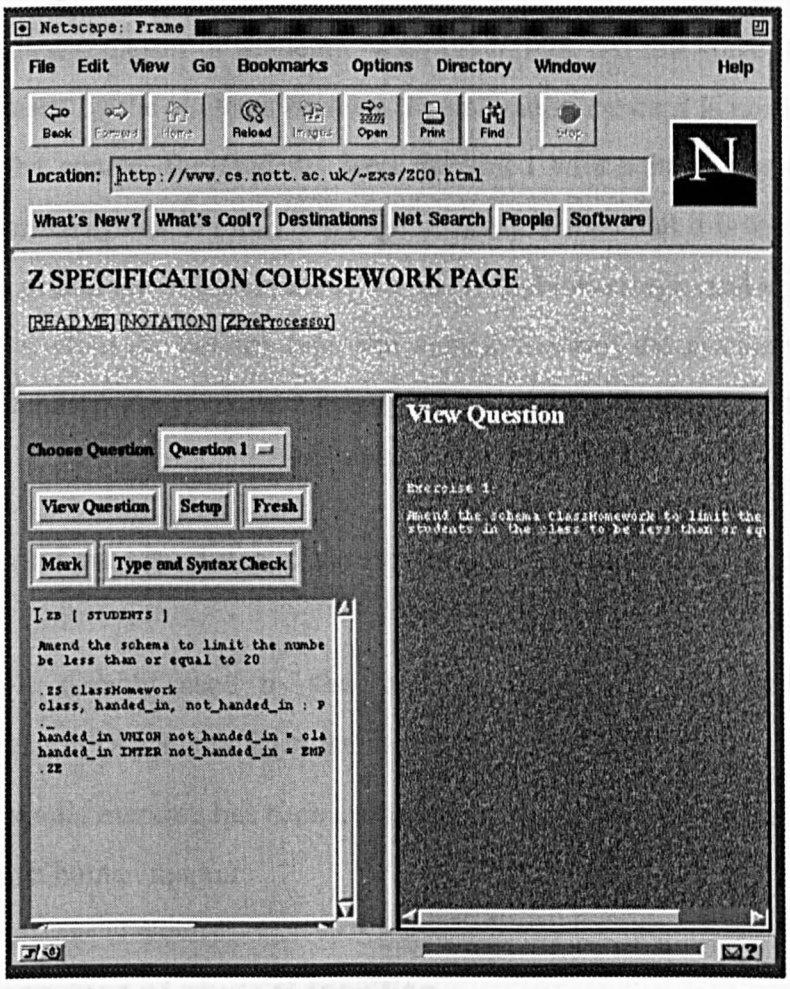


Figure 5-8 : Z Student Exercise Menu for WWW

The facilities include a text area to edit a Z schema in Z-roff format and six buttons with six different functions that are; to select a question, to view edited Z in PostScript form, to view the question, to retrieve the skeleton, to perform type and syntax checks and to mark the solution. The interface consists of three partitions. Related papers can be viewed from the top partition, the left partition is the work

area, and output from any action after clicking the buttons will be shown in the right partition.

5.6. Experience

The system has been tested on 13 final year students who took the Z specification course in the first semester of academic year 1997/1998. At this stage, the animator has not been used by the students because the animation concept is not in the course syllabus. In the course, the students were provided with a hardcopy of the `zpp` document. For the syntax of the Z language, it was assumed that it is the student who is responsible to find the information, as it follows first order logic and is well-defined in many books about Z. For the type and syntax checker, the grammar that it use follows the standard Z syntax and it only accept Z specifications written in `zpp` format.

The discussion of our experience of the system performance will be presented in two sections

- The tools directly used by the students have been evaluated by giving questionnaires (Appendix B) to the students.
- The automatic marking has been evaluated by comparing the system output with that from a human marker.

5.6.1. Performance of student facilities

At the end of the course, only 10 students filled in a questionnaire analysing the effectiveness of the system. The questionaire is divided into four sections:

1 Performance of the editor

2 Performance of the syntax and type checker

3 The need for an animator

4 General comments

Editor

The Z specifications are written using an ASCII text editor, with the format related to UNIX roff text processing. The PostScript form of the Z specification can be viewed using `ghostview`. 60% of the students have no experience in using the roff format, but reported that writing Z specification using it was not difficult. 90% of them preferred to write their Z specification in the roff format rather than by hand. Even though the roff format was reported as easy to use, all the students who already had experience of the WORDS mathematical library would prefer to use an editor specifically designed for editing Z specifications, which embeds the concept of WYSIWYG.

Type and syntax checker

In the system, the type and syntax checking is carried out by `zc`. 60% of the students found that `zc` was able to locate and explain some errors clearly. However, the students experienced that some errors detected by `zc` are not reasonable to their understanding. We found out that this happened due to the confusion in using `zpp` format. For example, the set subtract operator should be written as **SETSUB** in roff format, but many students wrote it as a symbol "-". When the type and syntax checker produce an error message saying that there is a type mismatch at the respective line, the student could not understand it. 70% of them claimed that it served a practical purpose and was useful. As expected, the students all claimed that there is a need for an ideal type and syntax checker in solving their coursework.

Animation

The concept of animation was not taught to the students. After explaining the concept to the students, 90% of them said that they would like to have a tool to animate their specification.

General comments

Below is a summary of the general comments from the students.

1 There is a need for on-line help in using the roff format.

2 There is a need for on-line help for the Z syntax.

3 There is a need for a better type and syntax checker.

One of the reason that the students asked for a better type and syntax checker might be because of the confusion in using Z-roff format. Although for the time being we do not have a Z editor, writing Z specifications in roff format is not a burden to the students.

From the student perspective, it can be concluded that teaching Z specification on-line will be more valuable if a Z editor, an ideal type and syntax checker and an animator are available.

From our perspective, the Z editor is feasible to be develop as it is only a matter of interface, whilst we can still use Z-roff as its backbone. For the type and syntax checker, we can use the current one as it serves the basic functions for the specification at the introductory level. For an animator, we found out that it is not suitable to be used by students at an introductory level. This is because the animation concept is not normally taught in the Z course.

Appendix C shows the feedback from the students.

5.6.2. Performance of the Z automatic marking system

At the moment we have tried the system with five Z exercises at an introductory level. The evaluation was done by comparing the results from the system with humans marking. By using a Correlation-Pearson test, we found that correlation between the system and the human is 0.837. This indicates a high level of correlation. The more detailed analysis is discussed in chapter 6.

5.7. Conclusion

In this chapter, we have described a system for the management of on-line Z specification coursework. The fundamental components of the process (the editor, the syntax and type checker, the animator and the automatic marking systems) were discussed and described as part of the well-known courseware management system, Ceilidh. Students can access the notes and exercises of the Z course on-line and submit their solution through Ceilidh. Finally the Z specification exercises are assessed automatically by the system. The focuses of this chapter is on the unique facilities that are provided by Ceilidh in handling Z specification exercises.

The system has been utilised by a group of 13 students who are taking a Z course. From the analysis, we show that the currently implemented tools are beneficial to the students. Preliminary testing of the automatic marking system reveals encouraging results.

A detailed study involving assessing more difficult exercises by the Z automatic marker and a range of human markers is the subject of following chapter.

References

1. S. Benford, E. Burke, E. Foxley, N. Gutteridge, and A.M. Zin, "Early experiences of computer aided assessment and administration when teaching computer programming," *Association for Learning Technology Journal*, vol. 1,

no. 2, pp. 55-70, 1993.

2. S. Hung, L. Kwok, and A. Chung, "New Metrics for Automated Programming Assessment," *IFIP Transactions A-Computer Science and Technology*, vol. 40, pp. 233-243, 1993.

3. M.J. Rees, "Automatic Assessment Aid for Pascal Programs," *SIGPLAN Notices*, vol. 17 , no. 10, pp. 33-42, October 1982.

4. P.B. Van Verth, "A System for Automatically Grading Program Quality," *SUNY (Buffalo) Technical Report*, 1985 .

5. K.A. Redish and W.F. Smyth, "Evaluating Measures of Program Quality," *The Computer Journal*, vol. 30, no. 3, 1987.

6. B. Woolf, "Intelligent tutoring systems: A Survey," in *Exploring artificial intelligence: Survey talks from the National Conferences on artificial intelligence*, ed. H. E. Shrode, pp. 1-41, Morgan Kaufmann Publishers Inc. , San Mateo, CA, 1988.

7. C. Youngblut, "Government-Sponsored Research and Development Efforts in the Area of Intelligent Tutoring Systems," (IDA Paper P-3003), Institute for Defense Analyses, Alexandria, VA. , September, 1994.

8. P. Brusilovsky and L. Pesin, "ISIS-Tutor: An adaptive hypertext learning environment," *Proc. JCKBSE'94, Japanese-CIS Symposium on knowledge-based software engineering.* , pp. 83-87, Tokyo, May 10-13, 1994.

9. A.M. Lesgold, S.P. Lajoie, M. Bunzo, and G. Eggan, "SHERLOCK: A coached practice environment for an electronics troubleshooting job," in *Computer assisted instruction and intelligent tutoring systems: Shared issues and complementary approaches* , ed. J. Larkin & R. Chabay , pp. 201-238, Lawrence Erlbaum Associates, Hillsdale, NJ, 1992.

10. E. Schwarz, P. Brusilovsky, and G. Weber , "World-wide intelligent textbooks. ," *Proceedings of ED-TELEKOM 96 - World Conference on Educational Telecommunications*, pp. 302-307, Charlottesville, VA: AACE., 1996.

11. S. Benford, E. Burke, E. Foxley, N. Gutteridge, and A.M. Zin, "Ceilidh as a Course Management Support System," *Journal of Educational Technology Systems*, vol. 22, no. 3, September 1993.

12. E. Foxley, E. Burke, C. Higgins, and C. Gibbon, "The Ceilidh System:A General Overview as at December 1996," *LTR Report*, University Of Nottingham, 1996.

13. R.S. Pressman, *Software Engineering*, McGraw-Hill Company Europe, 1992.

14. J.C.S. do Prado Leite and P.A. Freeman, "Requirements Validation Through Viewpoint Resolution," *IEEE Transactions on Software Engineering*, vol. 17, no. 12, pp. 1253-1269 , December 1991.

15. J.C. King, "Symbolic execution and Program testing," *Communication of the ACM*, vol. 19, pp. 385 - 394, July 1976.

16. R.A. Kemmerer, "Testing Formal Specifications to Detect Design Errors," *IEEE Transactions on Software Engineering*, vol. 11, no. 1, pp. 32-43, January 1985.

17. P. Jalote, "Testing the Completeness of Specification," *IEEE Transactions on Software Engineering*, vol. 15, no. 5, pp. 526-531, May 1989.

18. Steve Benford, Edmund Burke, and Eric Foxley, *Courseware to support the teaching of programming*, pp. 158-166, 1992. TLTP Conference, University of Kent at Canterbury

19. Z. Shukur, E. Burke, and E. Foxley, "Applying Z Specification Coursework On-Line," *Proceedings of AltC-98 Conference*, Oxford, UK, September 1998.

20. Z. Shukur, E. Burke, and E. Foxley, "Managing Z Specification Coursework On-line," *Journal of Computers in Mathematics and Science Teaching*, Charlottesville, USA, (submitted) 1999.

21. B.W. Kernighan and L.L. Cherry, "A System for Typesetting Mathematics," *Communications of the ACM*, vol. 18, pp. 151-157, 1975.

22. E. Foxley and A.M. Zin, *Zpp - A Troff Preprocessor for Typesetting Z Specifications*, 1990. Nottingham University Computer Science

23. A.M. Zin and E. Foxley, "Software Tools for Animating a Z Specification," *Sains Malaysiana*, vol. 24, no. 4, pp. 67-89, 1995.

24. A.M. Gravell, "What is a Good Formal Specification?," *Fifth Annual Z User Meeting*, Oxford, UK, 17 December 1990.

25. C. Jones, "A Survey of Programming Design and Specification Techniques," *Proceedings of IEEE Conference on Specification of Reliable Software*, pp. 91-103, Cambridge, Mass., 1979.

26. R. Macdonald, "Z Usage and Abusage," *Report 91003 Royal Signals and Radar Establishment*, Malvern, Worcs, February 91.

27. R. Balzer and N. Goldman, "Principles of Good Software Specification and their Implications for Specification Languages," *Proceedings of IEEE Conference on Specifications of Reliable Software*, pp. 58-67, Cambridge, Mass., 1979.

Chapter 6 : Evaluation

6.1. Introduction

We evaluate the performance of the automatic assessment based system described in this thesis by using samples from real students. Faidhi,[1] in his study on the complexity of Pascal programs said "all the programs collected are supposed to be fully working". Franklin[2] found in analysing the structure of Pascal programs automatically that before the system can take the sample to be used, the student had to make sure that the program (that they had written) should be correctly compiled, executed and seen to give correct output. Any failure meant that the students had to correct their program and re-submit it. The same applied to research carried out by Van Verth[3] on the automatic grading of program quality. She said, "each program was written in Pascal, was syntactically correct, produced correct results ...".

The method of evaluation of a system depends on the objective that we want to achieve. In this study, the evaluation that has been performed is quite different from the work described above. The main purpose of the evaluation of the AZAS system carried out in this chapter is to observe:

- the behaviour of the system compared with human markers, and
- the environment in which it can be used.

To achieve the first objective, participation from human markers were required. To achieve the second objective, it was decided that the experimental environment would not be bonded to the system requirements. The teacher and the student need not have any knowledge of the system. Basically, the steps taken in this study are:

- To determine the types of question that can be assessed by the system.
- To regard questions that are feasible for assessment by the system. The result from the system will be compared with result from human markers.
- To present the reasons for the questions which cannot be assessed by the system.

The result in this chapter has been included in the results section of a paper to IEEE Transactions on Software Engineering.[4]

6.2. Case studies

A group of 13 students in the School of Computer Science and Information Technology at the University of Nottingham took a Z specification module in the first semester of session 1997/1998. A set of Z courseworks had been implemented on-line, and the AZAS system was implemented within Ceilidh. The results from the automatic marker were not released to the students since this was an initial testing of the system and we felt that it should not be released to the students until it had been thoroughly tested. During the module each student had to do a number of courseworks which were obtained and submitted using Ceilidh. The courseworks were taken from books entitled *Logic and its Application*[5] and *Z: A Beginner's Guide.*[6] The students were informed that the system would mark their solution but the results would only be used for research purposes. It was stressed that their courseworks would be completely assessed by a human in the normal manner to obtain the final mark for their work.

6.3. Material selection

The questions selected are classified into three types.

- Questions that cannot be assessed by the system,
- Questions which can be assessed by the system directly, and
- Questions which need to be tailored before they can be used in the experiment.

The tailoring of the student answers will not affect the result of the experiment because the tailored answers will also be assessed by human markers. These experiments have increased our understanding about which is most suitable for AZAS. In general, it is not possible to assess certain questions by the system for the following reasons.

- No skeleton is attached.
- The question is not clearly defined.
- Some Z properties are not well enough supported by the Z to prolog translator.

Some editing of the student scripts was carried out for the following reasons.

- Problems were caused by inconsistencies in the specification of the question.
- Most of the students ignored the comments made by the type and syntax checker, `zc`. This was not surprising because the students were under no obligation take any notice of the comment given by `zc`.
- Most of the students modified the skeleton given to them. The vital modification was that they added some words to the schema name.

The compilation of the exercises is fully documented in Appendix D.

6.4. Data analysis

Sets of answers from ten exercises, (numbered accordingly) are used in this evaluation of AZAS. The exercises are marked by AZAS and the weight distribution is as follows:

5% each for typographics and complexity,
10% for static correctness, and
80% for dynamic correctness.

The same exercises are also marked by three different human markers who are represented by symbols h1, h2 and h3. The human markers were aware that they were marking the exercises specifically for the purposes of this experiment. The human marks were NOT the official University of Nottingham marks for these exercises. The official marks have NOT been released.

Data

As there are four different markers for the scores awarded, it can be said that the samples (i.e score) are collected from 4 independent groups. The *score* is in term of percentage, described as ratio/interval in statistics terms. The scores for every exercise from each of the markers are placed in a tabular form (like Table 6-1).

Score for Exercise 1				
Student	h1	h2	h3	system
s1	100	100	100	98
s2	100	100	100	99
s3	100	100	100	98
s4	100	100	100	99
s5	100	100	100	99
s6	100	100	100	98
s7	100	100	100	98
s8	100	100	100	98
s9	100	0	0	30
s10	100	100	100	99

Table 6-1 : Scores awarded by h1, h2, h3 and the AZAS system for Exercise 1

Even though the markers give a precise measurement, we can in this situation (refer to learning environment) classify this score to represent a rank such as A, B and so on, called the *grade*. The grade allows us to establish a rank ordering of answers from "good" answers to "poor" answers. In the University of Nottingham, the average of ALL modules is taken and a classification of degree awarded according to the following:

70%-100% : Class I

60%-69% : Class II division 1

50%-59% : Class II division 2

40%-49% : Class III

0%-39% : Fail

Because statistics deals with numbers, we associate the grade with a number as

follows:

Score which falls into 70-100% is associated with 5.

Score which falls into 60-69% is associated with 4.

Score which falls into 50-59% is associated with 3.

Score which falls into 40-49% is associated with 2.

Score which falls into 0-39% is associated 1.

The grades for every exercises are stored in a tabular form (like Table 6-2).

Grades for Exercise 1				
Student	h1	h2	h3	system
s1	5	5	5	5
s2	5	5	5	5
s3	5	5	5	5
s4	5	5	5	5
s5	5	5	5	5
s6	5	5	5	5
s7	5	5	5	5
s8	5	5	5	5
s9	5	1	1	1
s10	5	5	5	5

Table 6-2 : Grades awarded by h1, h2, h3 and the AZAS system for Exercise 1

It is also interesting to see the pattern of a frequency of the score regarding its classification for every exercise. Table 6-3 shows an example of how this information is stored in a tabular form for human marker h1. We refer to this data as a *grouped frequency.*

Exercise	70-100	60-69	50-59	40-49	0-39
1	10	0	0	0	0
2	10	1	0	0	0
3	11	0	0	0	0
4	8	2	1	0	0
5	11	0	0	0	0
6	6	0	3	1	0
7	7	1	1	1	0
8	2	3	3	0	2
9	12	0	0	0	1
10	10	1	2	0	0

Table 6-3 : Grouped frequency by h1

The scores, grade and grouped frequency for all the exercises awarded by the four markers are recorded in Appendix E.

Objective of the analysis

The data is analysed in order to compare the system marker behaviour and the human marker's behaviour in terms of:

the score,

the grade, and

the grouped frequency.

Plan for analysis

With the objective in mind, the plan of the analysis is presented as follows:

- Describe the raw data (i.e score) qualitatively. Any scores awarded by the markers that differ significantly will be discussed.
- Discuss the general outlook of the data.
- Identified the normality of the distribution of the data. This is important in selecting an appropriate test.

- Observe the correlation between the markers regarding the data.
- Test the hypothesis that *there is no difference between the system and human markers.*

6.5. Description of the score

Before exploring the statistical relevance of the results, we will present the raw data graphically. This is possible because the number of students to our study is relatively small. The scores awarded by the four markers for every exercise are shown using a barchart. Any visually extreme differences between the scores will be examined.

Exercise 1

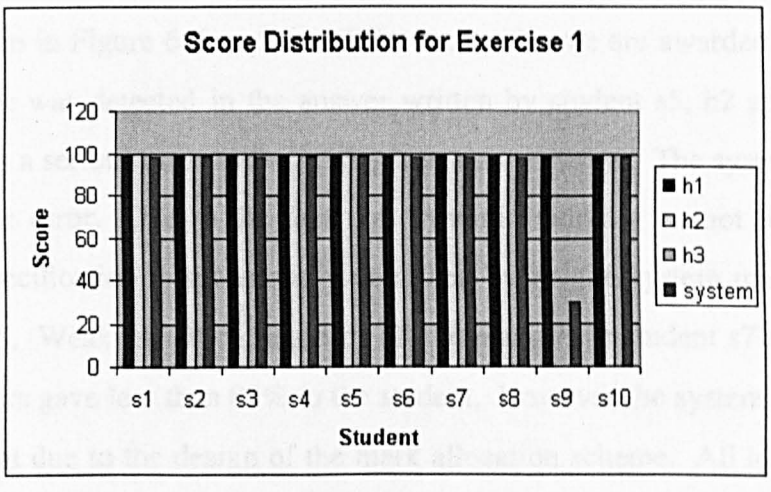


Figure 6-1 : Bar Chart of Score Distribution for Exercise 1

As shown in Figure 6-1, there are significant differences in the scores awarded for s9 by the markers. The automatic marking system detected that student s9 did not give appropriate predicates to specify the behaviour of the system in the question and so that student obtained a lower mark. h2 and h3 were also aware of the error which resulted in s9 obtaining no mark. Clearly, it can be seen that h1 was not careful enough to spot that the student answer was wrong.

Exercise 2

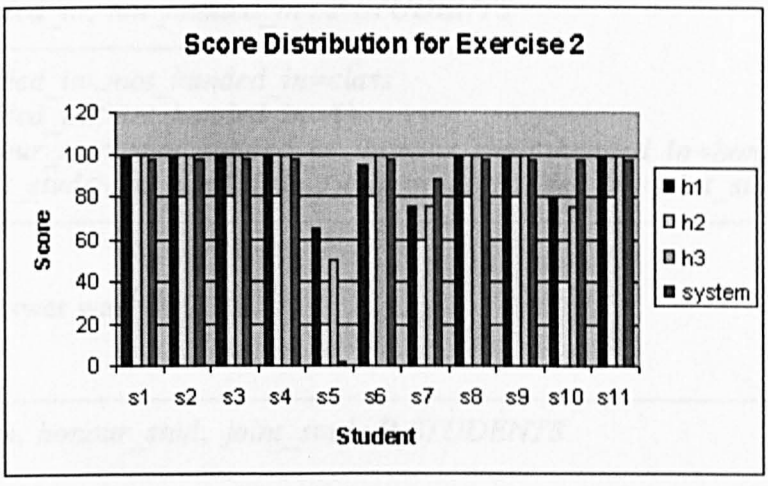


Figure 6-2 : Bar Chart of Score Distribution for Exercise 2

As can be seen in Figure 6-2, extreme differences of score are awarded to s5, s7 and s10. An error was detected in the answer written by student s5, h2 and the system classified it as a serious error as the mark given was very low. The system found it to be a syntactic error. Due to the fact that dynamic marking cannot be carried out unless the specification is successfully compiled by *zc*, the system marker awarded nearly 0 to s5. Weakness were found by all the markers in student s7's answer. All human markers gave less than 80% to the student. However the system gave a higher mark than that due to the design of the mark allocation scheme. All human markers were not satisfied with the answer given by s10. However, the system did not detect any weakness in the answer. The following examines the reason for this situation.

Observing the answer, it seems that the student presented the following schema:

Class

$$class, honour\_stud, joint\_stud : \mathbf{P}\ STUDENTS$$

$$honour\_stud \subseteq class$$
$$joint\_stud \subseteq class$$
$$honour\_stud \cap joint\_stud = \{\}$$

ClassHomework

$Class$
$handed\_in, not\_handed\_in : \mathbf{P}\ STUDENTS$

$handed\_in \cup not\_handed\_in = class$
$handed\_in \cap not\_handed\_in = \{\}$
$honour\_stud \cap not\_handed\_in \cup honour\_stud \cap handed\_in = honour\_stud$
$joint\_stud \cap not\_handed\_in \cup joint\_stud \cap handed\_in = joint\_stud$

The model answer was

Class

$class, honour\_stud, joint\_stud : \mathbf{P}\ STUDENTS$

$honour\_stud \cup joint\_stud = class$
$honour\_stud \cap joint\_stud = \{\}$

ClassHomework

$Class$
$handed\_in, not\_handed\_in : \mathbf{P}\ STUDENTS$

$handed\_in \cup not\_handed\_in = class$
$handed\_in \cap not\_handed\_in = \{\}$

Manipulation of the predicates is needed if one wants to observe the correctness of the answer. From the assessment made by the system (regarding the test cases provided), the answer is classified as correct. However, the maintainability factor surely should be considered in this case. It seems that the system is not sensitive enough to detect the weakness in the maintainability aspect.

Exercise 3

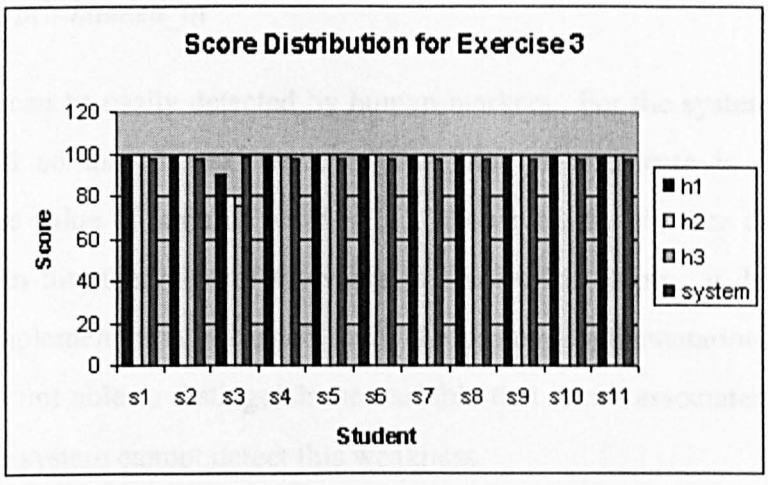


Figure 6-3 : Bar Chart of Score Distribution for Exercise 3

In Figure 6-3, only s3 obtains significant differences in the scores awarded. Human markers detect slight weakness in the answer given by s3. The student did not provide one predicate. The answer written by s3 is:

Add_Student

$\Delta ClassHomework$
$student? : STUDENTS$

$student? \notin class$
$class' = class \cup \{student?\}$
$not\_handed\_in' = not\_handed\_in \cup \{student?\}$

And the model solution is written as follows:

Add_Student

$\Delta ClassHomework$
$studs? : STUDENTS$

$studs? \notin class$
$class' = class \cup studs?$
$not\_handed\_in' = not\_handed\_in \cup studs?$
$handed\_in' = handed\_in$

The absence of the predicate

handed_in'=handed_in

in s3 answer can be easily detected by human markers. For the system, the test case was designed so that the existence of that kind of predicate is determined by evaluating the value of variable *handed_in'*. However, the absence of the predicate does not mean that the value of the variable involved is empty. It depends on how the prolog implementation is carried out. The prolog implementation being used in the system is not able to distinguish the variable that is not associated with a value. Therefore the system cannot detect this weakness.

Exercise 4

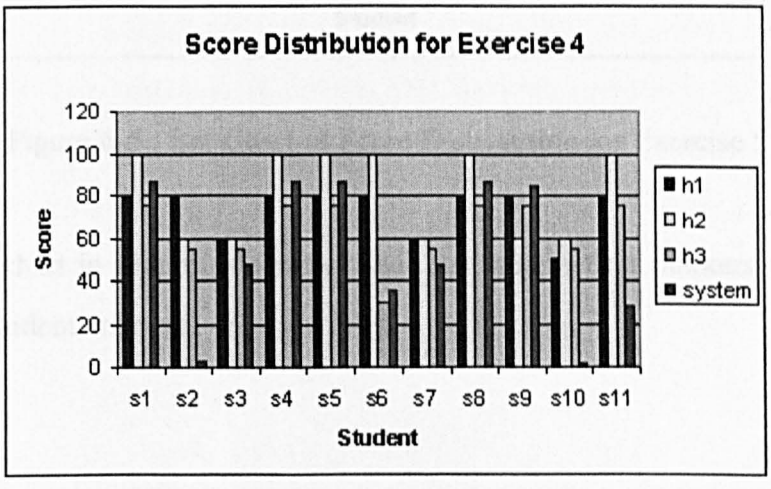


Figure 6-4 : Bar Chart of Score Distribution for Exercise 4

We set a more 'tricky' problem in exercise four, which caused several problems. The system and human markers detected this and marked the exercises accordingly. From Figure 6-4, we can see that the system alone is being more harsh to students s2, s10 and s11. We found that answers from those two students are essentially correct, but they had problems with the type and the syntax (which can be put down to the carelessness of the student). They could therefore not get any marks from the

dynamic marking and complexity aspects of the system. For s11, it is statically correct, however, it does not satisfy most of the test cases during dynamic correctness.

Exercise 5

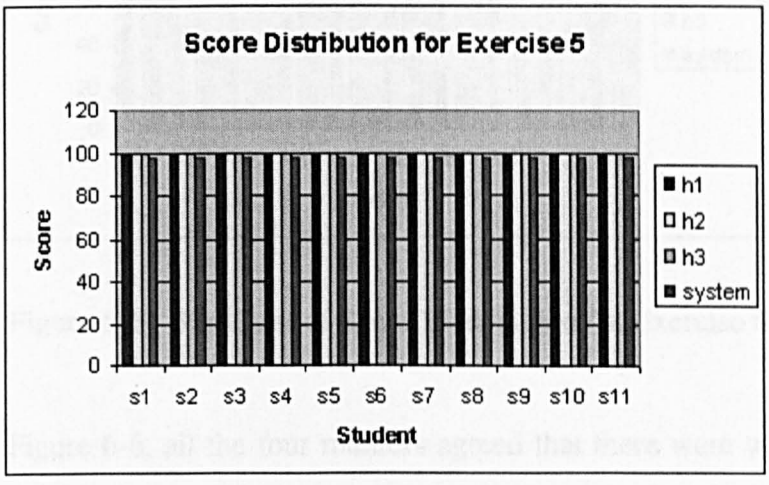


Figure 6-5 : Bar Chart of Score Distribution for Exercise 5

From the barchart in Figure 6-5, we can see that all the four markers were satisfied with all the student answers.

Exercise 6

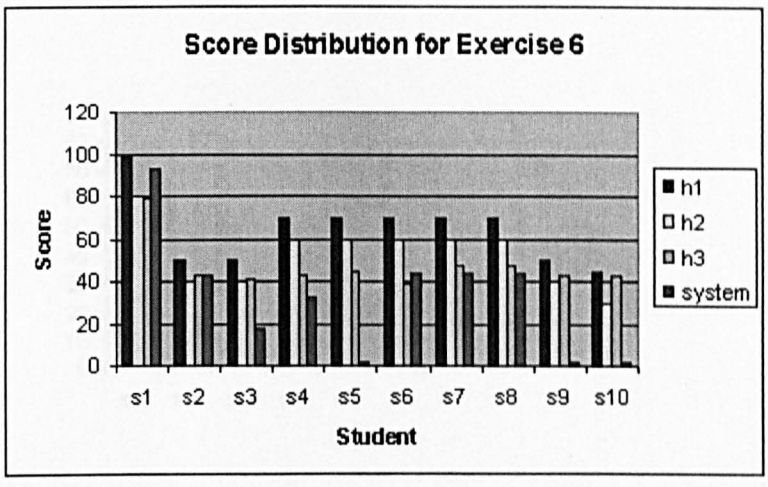


Figure 6-6 : Bar Chart of Score Distribution for Exercise 6

As shown in Figure 6-6, all the four markers agreed that there were weakness in all the student solutions, with s3, s5, s9 and s10 showing significant differences in the score awarded by the markers. The system is more harsh to s5, s9 and s10. Again, this is due to syntactic problems in the student answers. For s3, the answer was syntactically correct. However it failed all of the analysis during the dynamic marking.

Exercise 7

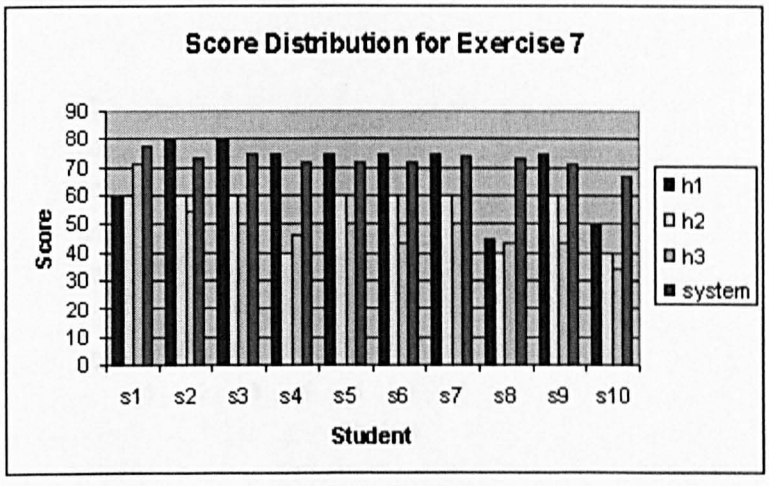


Figure 6-7 : Bar Chart of Score Distribution for Exercise 7

At a glance, the pattern in Figure 6-7 shows that the system behaves almost like h1. There are visually significant differences in the score given to s8 and s10. The human markers detect an error in both of the student answers. Looking at the score given, they categorised it as serious. Even though the system detected the weakness, the score awarded is quite high due to the mark allocation scheme's design.

Exercise 8

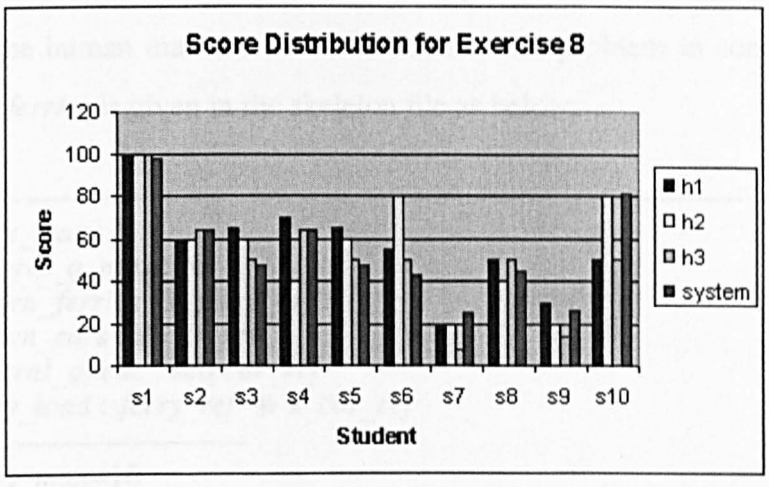


Figure 6-8 : Bar Chart of Score Distribution for Exercise 8

In general, the system tends to behave like the human markers. Even though the system gave s10 high score, this is also done by h2. In fact, we can also see that h2 did not agree with other markers for s6.

Exercise 9

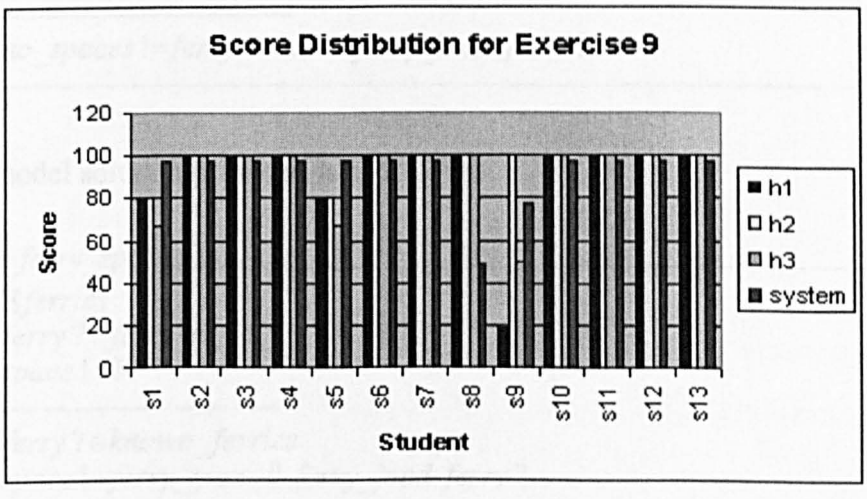


Figure 6-9 : Bar Chart of Score Distribution for Exercise 9

Extreme differences can be spotted in s1, s5, s8 and s9. From the score given to s1 and s5, we can see that the system did not agree with the weaknesses that were detected by the human markers. We will look at this problem in some detail. The state schema *ferries* is given in the skeleton file as below:

ferries

$$
\begin{aligned}
&ferry\_max : \mathbf{N} \\
&general\_q\_max : \mathbf{N} \\
&known\_ferries : \mathbf{P}\ ferry\_ref \\
&known\_cars : \mathbf{P}\ car\_ref \\
&general\_queue : \mathbf{seq}\ car\_ref \\
&ferry\_load : ferry\_ref \pfun \mathbf{P}\ car\_ref \\
\hline
&ferry\_max = 10 \\
&general\_q\_max = 20 \\
&known\_ferries = \mathbf{dom}\ ferry\_load \\
&known\_cars = \mathbf{rng}\ general\_queue \cup \bigcup \mathbf{rng}\ ferry\_load \\
&\#\ general\_queue \leq general\_q\_max \\
&\forall f : known\_ferries \bullet \#\ ferry\_load \leq ferry\_max
\end{aligned}
$$

Both of students s1 and s5 answers are as follows:

list_ferry_spaces

$$
\begin{aligned}
&\Xi ferries \\
&ferry? : ferry\_ref \\
&no\_spaces! : \mathbf{N} \\
\hline
&no\_spaces! = ferry\_max - \#\ ferry\_load\ ferry?
\end{aligned}
$$

And the model solution provided is as follows:

list_ferry_spaces

$$
\begin{aligned}
&\Xi ferries \\
&ferry? : ferry\_ref \\
&space! : \mathbf{N} \\
\hline
&ferry? \in known\_ferries \\
&space! = ferry\_max - \#\ ferry\_load\ ferry?
\end{aligned}
$$

The human markers awarded less marks because the student answer did not provide the pre-condition

$ferry? \in known\_ferries$

However, the system understands this differently. The function # *ferry_load ferry*? in the predicate part of the student's schema is dynamically wrong if *ferry*? does not exist in the domain of *ferry_load*, and otherwise. Therefore the system awards nearly full marks.

It was a totally different situation for student s8. All the three human markers agreed that there is no fault with the student answer, whilst the system detected an error. Consider student s8 answer

list_ferry_spaces

$\Xi ferries$
$ferry? : ferry\_ref$
$result! : \mathbf{N}$

$ferry? \in ferry\_ref$
$result! = ferry\_max - \#\ ferry\_load\ ferry?$

At a glance, it is difficult for a human to detect the error. However, if we look carefully, the predicate

$ferry? \in ferry\_ref$

is actually the problem. According to the model solution, it should be

$ferry? \in known\_ferries$

ferry_ref is an object type whilst ***known_ferries*** is a system variable.

We can also see that significantly different classes of marks was awarded to student s9, where h1 give very low marks, h2 awards full marks while the system and h3 are less extreme. The answer written by s9 is:

list_ferry_spaces

$\Xi ferries$
$ferry? : ferry\_ref$
$spaces! : \mathbf{N}$

$ferry? \in known\_ferries$
$spaces! = \#\ ferry\_load\ ferry? - ferry\_max$

The problem is that variable *spaces*! will yield a negative value of the correct answer. From the marks awarded, we can assume that h1 classified it as a serious error, whereas h2 might not be aware about the error and h3 is quite understanding about it. The system detected the error, and awarded marks according to the mark allocation scheme.

Exercise 10

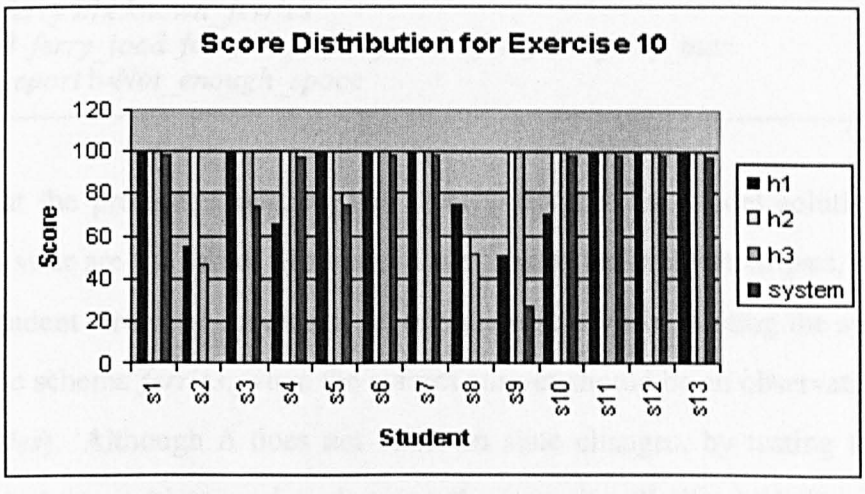


Figure 6-10 : Bar Chart of Score Distribution for Exercise 10

Figure 6-10 shows that s2, s3 and s8 experienced significant differences in the score awarded. Again, the human markers agreed that there is no fault in specification written by s3. However, the system detected something. Observe the student answer which is as follows:

can_ferries_combine_no_space

$\Delta ferries$
ferry_from?: *ferry_ref*
ferry_to?: *ferry_ref*
result!: *message*

---

ferry_from?$\in$*known_ferries*
ferry_to?$\in$*known_ferries*
ferry_load ferry_from?+# *ferry_load ferry_to*?>*ferry_max*
result!=*Not_enough_space*

And the model solution is as follows:

can_ferries_combine_no_space

$\Xi ferries$
ferry 1?: *ferry_ref*
ferry 2?: *ferry_ref*
report!: *message*

---

ferry 1?$\in$*known_ferries*
ferry 2?$\in$*known_ferries*
ferry_load ferry 1?+# *ferry_load ferry* 2?>*ferry_max*
report!=*Not_enough_space*

Looking at the predicates part we can clearly see that the model solution and the student answer are the same. However if we look at the declaration part, we can see that the student wrote the schema as an operation schema by writing the symbol Δ in front of the schema *ferries*, when the correct answer should be an observation schema (i.e Ξ*ferries*). Although Δ does not insist on state changes, by testing the schema against a test case which involve changing the state, it will allow this happen, which it should not.

In this exercise, the answer written by s2 was decided by h1 and h3 to be not good enough. However, h2 decided to give full marks. The system awarded 0 marks because the process of marking was interrupted due to 'hanging' during the run-time. As well as the answer written by s8, all the markers were in agreement that there was a fault in the specification. The system gave 0 marks due to the same reason as s2. 'Hanging' during run-time is a problem in the Prolog implementation of the

respective specification. The 'hanging' is due to the uncontrolled back tracking in the implementation. This type of problem is discussed in chapter 7, and further recommendations are also presented in the chapter.

6.6. Inferential statistics

Studies concerned with infering conclusions about populations based on results obtained from samples (by using certain procedures) is known as inferential statistics. The following are analyses of the samples by using selected procedures appropriate to the nature of the samples. SPSS tool version 8 was used in this analysis.

General outlook

To have a general overview of the data, we look at its mean. The mean values for the score, grade and grouped frequency awarded by the markers are shown in Table 6-4.

Marker	Score	Grade	Frequency
H1	83.73	4.58	2.20
H2	81.00	4.37	2.20
H3	78.05	4.10	2.20
SYSTEM	75.85	4.09	2.16

Table 6-4 : Means for the distribution

As displayed by the table, the system mean is the lowest compared to the human markers for all the 3 types of data. Are these differences significant? At this stage, we are not able to conclude anything. We will continue to analyse this (below).

Identification of the normality

It is important to know the distribution of the data, as it will guide us in selecting an appropriate test. A quantile-quantile (qq) plot is used to see if a given set of data follows a specified distribution. It should be approximately linear if the specificied

distribution is the correct model.

In our study, we are going to observe whether our data are normally distributed. We use h1's distribution of score as an example of how to determine the normality. As revealed by the graph in Figure 6-11, the distribution of the data is not in a straight line. To have a clear picture of how the data deviates from the normal, we can explore its detrended normal plot. If the sample is from a normal distribution, the points should cluster in a horizontal band around zero; there should not be a pattern. In this example, the points were not clustered around zero (which can be seen in Figure 6-12). Therefore we can conclude that the score distribution by human marker h1 is not normal.

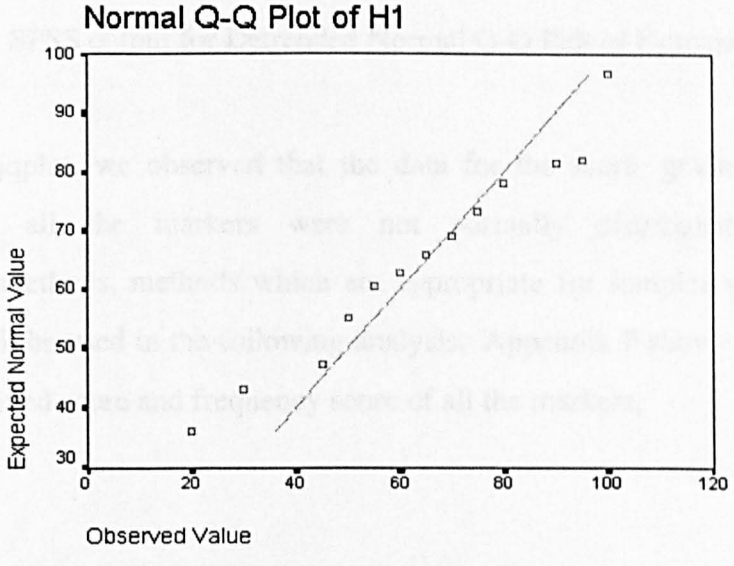


Figure 6-11 : SPSS output for Normal Q-Q Plot of Human Marker H1

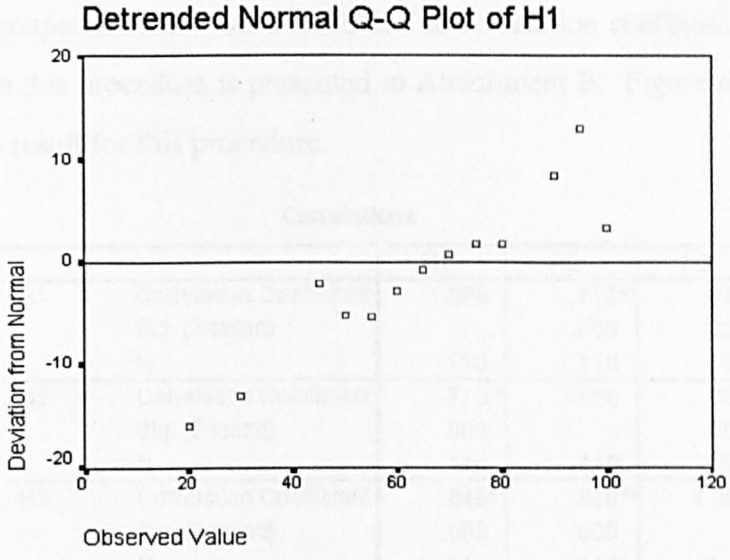


Figure 6-12 : SPSS output for Detrended Normal Q-Q Plot of Human Marker H1

By using the qqplot, we observed that the data for the score, grade and grouped frequency for all the markers were not normally distributed. Therefore, nonparametric methods, methods which are appropriate for samples with unknown distribution, will be used in the following analysis. Appendix F shows the qqplot for the score, classified score and frequency score of all the markers.

Correlation

Correlation is a study of the relationship between two variables. The measurement for correlation is called the correlation coefficient. In this section, we would like to observe the correlation between the system and the humans in terms of the distributed score, grade and grouped frequency. Additionally, we would also like to study this correlation relative to the correlation between human and human. In the nonparametric method, there are two ways to measure the association of variables; using the Spearman correlation coefficient and the Kendall Tau correlation coefficient. In most of the literature, it is said that both tests can be used. I was verbally advised by one researcher majoring in statistics to use the Spearman

correlation coefficient for analysing the data score because it is ratio/interval typed. For grade and grouped frequency, the Kendall Tau correlation coefficient was chosen. The data used in this procedure is presented in Attachment E. Figure 6-13, 6-14 and 6-15 display the result for this procedure.

Correlations

			H1	H2	H3	SYSTEM
Spearman's rho	H1	Correlation Coefficient	1.000	.713**	.845**	.767**
		Sig. (2-tailed)	.	.000	.000	.000
		N	110	110	110	110
	H2	Correlation Coefficient	.713**	1.000	.806**	.676**
		Sig. (2-tailed)	.000	.	.000	.000
		N	110	110	110	110
	H3	Correlation Coefficient	.845**	.806**	1.000	.826**
		Sig. (2-tailed)	.000	.000	.	.000
		N	110	110	110	110
	SYSTEM	Correlation Coefficient	.767**	.676**	.826**	1.000
		Sig. (2-tailed)	.000	.000	.000	.
		N	110	110	110	110

**. Correlation is significant at the .01 level (2-tailed).

Figure 6-13 : SPSS output for Spearman's Correlation among markers in terms of Score distribution

Correlations

Figure 6-14 : SPSS output for Kendall Tau's Correlation among markers in terms of Grade distribution

Correlations

Figure 6-15 : SPSS output for Kendall Tau's Correlation among markers in terms of Grouped Frequency distribution

As indicated in Figures 6-13, 6-14, 6-15, the correlations among the markers are very strong and all achieve a high level of statistical significance. We can conclude that the correlation between the system and the 3 humans is significant in terms of score,

grade and grouped frequency.

The variety of coefficients in the figures can be grouped into two; the coefficient between system and human, and the coefficient between human and human. We are interested to see whether the differences between system-human correlation and human-human correlation (regardless of which marker) is significant. We hypothesize that *there is no difference between the correlation between system and human and the correlation among humans.* Table 6-5 is the summary of the correlation value from the score, grade and grouped frequency. As correlation coefficients are special data which were derived from the correlation procedure, there might be a special test for them, such as an extension of the correlation procedure. However, from several literature reviews about testing the differences between correlation coefficients, none was found to be suitable. The closest is testing for two correlation coefficients. Therefore, we decided to treat the coefficients as data with unknown distribution.

Correlation					
Score		Grade		Frequency	
system-human	human-human	system-human	human-human	system-human	human-human
0.676	0.713	0.545	0.476	0.473	0.424
0.767	0.806	0.582	0.525	0.490	0.563
0.826	0.845	0.666	0.741	0.593	0.582

Table 6-5 : Summary of Correlation Coefficient

The Mann-Whitney-Wilcoxon test is appopriate for this case, as it is a nonparametric test for two independent samples. The null hypothesis for this test is that:

> there is no difference between correlation coefficient system-human and human-human.

The alternative hypothesis is that:

> there is a difference between correlation coefficient system-human and human-human.

We ran the test for the 3 types of data taken from Table 6-5. The results of the tests are presented in Figure 6-16, 6-17 and 6-18.

Test Statistics[b]

	correlation
Mann-Whitney U	3.000
Wilcoxon W	9.000
Z	-.655
Asymp. Sig. (2-tailed)	.513
Exact Sig. [2*(1-tailed Sig.)]	.700[a]

a. Not corrected for ties.

b. Grouping Variable: MARKER

Figure 6-16 : SPSS output for the Mann-Whitney test comparing correlation coefficient for Score distribution between system-human correlation and human-human correlation

Test Statistics[b]

	correlation
Mann-Whitney U	3.000
Wilcoxon W	9.000
Z	-.655
Asymp. Sig. (2-tailed)	.513
Exact Sig. [2*(1-tailed Sig.)]	.700[a]

a. Not corrected for ties.

b. Grouping Variable: MARKER

Figure 6-17 : SPSS output for the Mann-Whitney test comparing correlation coefficient for Grade distribution between system-human correlation and human-human correlation

Test Statistics[b]

	correlation
Mann-Whitney U	4.000
Wilcoxon W	10.000
Z	-.218
Asymp. Sig. (2-tailed)	.827
Exact Sig. [2*(1-tailed Sig.)]	1.000[a]

a. Not corrected for ties.

b. Grouping Variable: MARKER

Figure 6-18 : SPSS output for the Mann-Whitney test comparing correlation coefficient for Grouped Frequency distribution between system-human correlation and human-human correlation

The figures show that the p-values for score, grade and grouped frequency (which are 0.513, 0.513 and 0.827) are not significant based on the confidence of 95%. It means that there is not enough evidence to reject the null hypothesis. In other words, the relation between system-human can be regarded as the same as the relation between human-human.

Testing a hypothesis

Even though we have explored the data thoroughly, and confidence about the correlation between system and human is as strong as the correlation between human and human, we have not answered our early hypothesis of this analysis, that is, *there is no difference between score, grade or grouped frequency produced by the system and by the human markers.*

According to the nature of the data discussed before, an appropriate statistical test to carry out in this case is the Kruskal-Wallis test for K independent samples. The Kruskal-Wallis test is a nonparametric test suitable for more than two independent samples where the samples come from unknown distribution.

The raw data (i.e score), is not suitable for this analysis. because nonparametric analysis will rank the data, and scores like 100% and 97% cannot really be ranked as best or second best. However, grade and frequency are appropriate for this test.

The null hypothesis for both data is:

there is no difference between grade/grouped frequency of different markers.

The alternative hypothesis is that:

there is a difference between at least two of the markers.

The data used in this test is presented in Attachment E. The results from the test for grade and grouped frequency are shown as in Figure 6-19 and 6-20, respectively.

Test Statistics[a,b]

	GRADE
Chi-Square	7.210
df	3
Asymp. Sig.	.065

a. Kruskal Wallis Test

b. Grouping Variable: MARKER

Figure 6-19 : SPSS output for the Kruskal-Wallis test comparing Grade awarded by the markers

Test Statistics[a,b]

	FREQUENC
Chi-Square	.221
df	3
Asymp. Sig.	.974

a. Kruskal Wallis Test

b. Grouping Variable: MARKER

Figure 6-20 :SPSS output for the Kruskal-Wallis test comparing Grouped Frequency awarded by the markers

Based on the p-values of 0.065 for grade and 0.974 for grouped frequency (which are not significant at the level of 0.05), this implies that there is not enough evidence for us to reject the null hypothesis. This means that we can say that the grade and

grouped frequency given by the system is the same as that given by a human.

6.7. Observations

Bearing in mind the first objective of the experiment which is to observe the behaviour of the system compared to human markers, we can summarise as follows:

- The system marker is better at detecting certain type of errors which a human marker can often overlook.
- Human markers are better at detecting certain type of erros which a system marker could not detect regarding the insensitivity of the test cases.
- When dealing with a serious error, the human markers are more lenient in awarding marks compared to the system marker.
- The system marker has a strong correlation with human markers in terms of determining score, grade and grouped frequency.
- The correlation concerning score, grade and grouped frequency between the system and humans is believed to be the same that between human and human.
- The system grades for the student answers are similar to the human grades.
- The system groups of the grade of student answers are similar to the human groups.

Regarding the second objective which is to observe the environment in which it can be used, we can conclude that the system can only be used for Ceilidh-type exercises. This means:

- The skeleton should be attached.
- The question must be clearly defined.
- The schema name, the type of input and output must be determined by the teacher.

From the data analysis, we also found that:

- There is a need for technique to allocate marks for test cases. This is to ensure that the marks awarded will be appropriately distributed according to the seriousness of the error. We propose such technique in the next chapter.
- If the system is to be used to produce official students' results, the use of the type and syntax checker by the student should be emphasised. The students should be informed that any syntactical error will yield substantial loss of marks.
- The Prolog implementation of Z needs improvement and it needs to be expanded to support other functions. This matter will be further discussed in the next chapter.

6.8. Conclusion

In this chapter, we have presented and discussed an experiment to observe and compare the performance of the Automatic Z Specification Assessment System with human markers. Samples are taken originally from students and used in the analysis. By using decriptive and inferential statistics, we conclude that the system indeed behaves like a human marker in some areas and indeed in certain areas it is better than a human. As the conclusions drawn above are only valid for the tested sets of answers, there is a need for more experiments to be conducted before substantial claims concerning the performance of the system can be made. A final remark: as the number of participants is quite small (i.e 13), the strength of the statistical result can be argued. However, this situation cannot be avoided as the module is optional and few students take it. We have chosen the best statistical test that exists to suite the nature of our sample.

References

1. J.A.W. Faidhi, "The complexity analysis of Pascal programs and the application to a university teaching environment," *PhD Thesis*, University of Brunel, 1986.
2. M.E. Franklin, "Automatic analysis of the structure of Pascal programs written by novices," *MPhil Thesis*, University of Sheffield, 1987.
3. P.B. Van Verth, "A System for Automatically Grading Program Quality," *SUNY (Buffalo) Technical Report*, 1985 .
4. Z. Shukur, E. Burke, and E. Foxley, "Inspecting the correctness of specification through system-state analysis," *IEEE transaction on Software Engineering*, (will be submitted) 1999.
5. E. Burke and E. Foxley, *Logic and its Applications*, Prentice Hall Europe, 1996.
6. D. Rann, J. Turner, and J. Whitworth, *Z: A Beginner's Guide*, Chapman & Hall, 1994.

Chapter 7
Discussion and Further Research

7.1. Introduction

This chapter summarises the contributions to be found in this thesis. It also makes a critical assessment of the research area as a whole and presents possible directions for future research.

Using the marking system within Ceilidh could not be easier. The interface provided by Ceilidh, makes it easier for the teacher to set up the exercise and marking scheme. The teacher need not to understand how the marking system works in detail. It is enough for them to understand the basic functions of Ceilidh.

7.2. Contributions

In this thesis we proposed a system to automatically assess student Z specification exercises.

In the first chapter of the thesis, we discussed the need to have an automatic marking system for a specification in a learning environment. We also briefly discussed why Z is chosen in the study. By presenting a relation between a program and a specification in chapter two, we arrive at a point where ideas in analysing the program automatically can be used in analysing a specification. In general, four aspects should be considered if one wants to develop such a system. They are:

- Identify factors that affect the quality of a subject to be assessed.
- Regarding the identified factors, find a way to derive the quality metric from the subject.
- Find a quality indicator to be the milestone of the quality.
- Select or derive a suitable scheme to award the mark.

The system that we developed considers four quality factors when assessing Z specifications; typographics, complexity, static correctness and dynamic correctness.

In chapter three, we described a system to automatically assess the quality of Z specifications by considering maintainability and correctness. The analysis of a specification's correctness is done in two steps: first the specification will be statically analysed by applying a type and syntax checker against it. Second, if no error occurs then the analysis will continue by checking its dynamic correctness. This is carried out by animating the specification against a set of test data. The analysis of the maintainability of a specification is divided into two aspects: analysis of the typographics arrangement of a specification, and the analysis of the complexity of a specification.

The dynamic correctness employs the testing technique. In chapter four we discussed how the idea is handled regarding the context of specification. By using a system-state analysis approach, the specification is 'executed' against sets of test data. System variables are used as the test data. After the specification is invoked, the system variables (i.e variables which hold the value of the system-state, as well as input and output variables) are checked. The result of the testing is compared to the expected result provided by the teacher. The comparisons yield a value in terms of a percentage. If the result is less than 100%, the cause of the lost marks can be discovered. If an error is revealed, the debugging process can be invoked to locate the error. In chapter four, we also proposed a way to derive test data together with a scheme for awarding marks.

The need for basic tools for handling Z coursework on-line is highlighted in chapter five. The incorporation of the automatic marking tools along with the existing editor, type and syntax checker and animator in Ceilidh are explained. We conducted an empirical analysis on student views about the facilities offered in the system. The results indicated that this could be very beneficial to the student. However, we found out that the need for better tools are strongly proposed by the students.

To observe the performance of the marking system, an experiment was conducted where specifications submitted by students at the introductory level were used. The selection of the material is discussed. Comparisons with marks given by human markers for the same sets of specifications have demonstrated that the system marker tends to behave like a human marker. This is presented in chapter six.

7.3. Outstanding problems

Even though we have shown the possibility of marking a Z specification automatically, the techniques currently used restrict its area of application in a number of ways. The problems are divided into two main areas; local problems (i.e the problems which contribute to the shortcomings in the research itself) and universal problems (i.e the problems which are also being discussed by other researchers).

7.3.1. Local problems

We identify some weaknesses that are encountered in the study.

- The handling of dynamic correctness is designed specifically for Ceilidh-type exercises. This means that the schemas involved will be determined by the teacher and the input and output will be set by the teacher. Therefore this analysis could not be applied to a modelling-type exercise.
- The idea of awarding a mark for the failed schema when it should be true is not absolute. Giving marks according to the percentage of predicates which contribute to the correctness yields a problem. If the location of the predicate that contributes to the failure is changed to the other line, the system might award different marks.
- Some bugs in *zpp* need to be removed.[1]
- The Z to Prolog translator *zp* needs to support more features of Z specification techniques.[1]

During the study, we corrected some of the bugs that were found. In Appendix G we highlight some deficiencies that were found in the tools.

7.3.2. Universal problems

There are some universal problems that are still being discussed among the researchers in this area.

Prolog as a media for animation

We chose Prolog as a suitable language for the animation of Z specifications because they are both based on first order logic (and set theory). Several attempts have been made to translate Z schemas to Prolog code using different methods. However, this is said to have had little impact.[2] Prolog code that has been produced has often been very inefficient due to inferior transformation rather than to limitations of the approach.[2] As an example, in the SuZan project,[3] they faced two main problems.

- Combinatorial explosion (i.e no satisfactory answer will be obtained by a query either because there are infinite answers or because the generate and test loop will take too much time).
- Limited translation for schema calculus.

Another example is EZ.[4] With EZ, only a subset of Z can be translated. Further, it did not support

- $\exists$ and $\forall$
- axiomatic declarations
- generic schemata
- set and multiset operations
- functions and relations
- schema operator $\rangle\rangle$

- a few logical operators like $\Leftrightarrow$ and $\Rightarrow$

The predicate library in *tzc*[2] (a compiler to translate Z schemas into Prolog code), needs to be augmented in order to handle the full range of Z predicates and expressions.

Sterling et al[2] commented (on Prolog) that "things like types tend to be defined implicitly: a type is whatever it is". Whereas the distinctive features of Z is that it is a typed language.[5] They further said "this would make hard to tie down certain properties about functions; for example, whether a function is total or partial becomes more of a philosophical issue than a practical one".

There have also been attempts to translate formal specification into functional languages.[6,7] However, it is stated by Gravell and Henderson[8] that "the translation scheme was not automated".

Animating a specification

As we have described earlier, the system uses testing techniques to validate a specification. The only way to do this is to make it executable. Execution of a formal specification is said by Gravell & Henderson[9] to be a controversial area. The debate[10,11] on these issues will not stop and, of course, it is not the objective of this thesis to join the debate. Gravell and Handerson[9] summarised the debate as follows:

- There is a conflict between clarity and executability.
- There are relative merits of proof and execution in validating specifications.
- There is a potential unreliability of manual transliteration for execution.
- There is an inefficiency of some forms of execution
- There are possible merits of hybrid forms of execution.

In the following section, we will investigate other possible ways that can be used in the system to check the correctness of Z specifications.

7.4. Direction for future research

7.4.1. Refinements to the system

One of the areas of research that was beyond the scope of this project was an exhaustive investigation of specification maintainability. This might be because the environment that we are working on is learning, where the factor of correctness is heavily considered rather than maintainability. However, in a real situation, maintainability is identified as being as important as other quality factors. Therefore we propose:

- Investigation of the typographics arrangement for Z specifications, by considering factors which are specifically for a Z specification. This was not taken into consideration during the research because of the difficulties in automation. Such a problem is another major research project in itself which would fall in the area of Artificial Intelligence. In addition, more work could be carried out in choosing a quality indicator for typographics assessment. No clear solutions have been proposed from the literature to address this problem. Some work could be carried out in the further evaluation of the effectiveness of the typographics factors.
- Specification complexity factors are not precisely debated by scholars in the area of software metrics and specification. Investigation of these factors should be conducted. The aim of the study might be to investigate complexity in Z specifications and to find a practical way to derive the complexity metric. Most of the complexity factors involve the analysis of a specification as a whole. Therefore the effectiveness study of these factors might be carried out independently.

Some other areas of future research which are expected to be short term projects are:

- Improve the student facility to test their specification against the test data. This will present a detailed analysis of the failure.

- Provide a facility for students to view the question in PostScript form. This was actually done at an early stage of the research. However it was discontinued because of the time to load the PostScript viewer. The main thing that has been considered during this project is response time.
- Provide a facility for the students so that they can test their specification using their own initial data state and check the final data state without having to understand the animation concept.
- There are lots of problems which can occur when directly using the Unix spell checker. There are some vocabulary which is not in the Unix System Vocabulary. Examples are vocabulary of the complete course notes, the course title, vocabulary of the question, the test data, student's program identifiers and other such vocabulary. This problem has been resolved by the Ceilidh spell checker.[12] Therefore instead of using the Unix spell checker in AZAS, invoke the Ceilidh spell checker.

7.4.2. Correctness analysis using symbolic execution

With the development of a reliable theorem prover, symbolic execution is one of the techniques which we might use (apart from animation). For example, the student might be asked to write a schema operation named *AddTelephone* where a person's telephone number will be added into the telephone database provided that it is not set in the database.

AddTelephone

$$\Delta TelephoneBook$$
$$name? : NAME$$
$$phone? : TELEPHONE$$

$$name? \notin known$$
$$known' = known \cup \{name?\}$$
$$telephone' = telephone \cup \{(name?, phone?)\}$$

The sequence of starting a *TelephoneBook*, then adding a person's telephone number

can be tested by defining the schema

$$TestTelephone == InitTelephone \ ; AddTelephone$$

and then expanding the schema *TestTelephone*. *InitTelephone* is a schema which initialises the database with empty data. The result of symbolically executing it will be something like this:

$known = \{\}$

$telephone = \{\}$

$name? \in NAME$

$phone? \in TELEPHONE$

$name? \notin known$

$known' = \{name?\}$

$telephone' = \{(name?, phone?)\}$

The above result presents symbolic values for the system variables. This result can be compared to the model result which is provided by the teacher.

7.4.3. Correctness analysis using formal proof

Instead of using a testing technique, we would like to show that a formal proof might possibly be used in validating and verifying a specification. We describe the process by giving the same example as that presented previously (i.e *AddTelephone*). In the example, it is obvious that after adding the telephone number of a given name, the respective name and the telephone number should be recorded in the telephone database. By using an existing theorem prover (such as Isabelle[13] and Z/EVES[14]), this can be automatically checked by proving the following proposition.

$$\forall AddTelephone \bullet name? \notin known \land telephone'(name?) = phone?$$

To suite our problem, we might prove this separately and allocate a mark for each of the following steps.

if $\forall AddTelephone \bullet name? \notin known$ is *true* then give 5 marks.

if $\forall AddTelephone \bullet telephone'(name?)=phone?$ is *true* then give 10 marks.

7.4.4. Generation of a test case for marking purposes

A lot of research has been carried out to generate test cases from a formal specification. Here we will look at some techniques which focus particularly on generating test cases from a Z specification.

Hall[15] proposed an approach for generating test cases from Z specifications. This was done by first identifying the test domains. Having established the test-domains, "typical" elements were selected from the sets. The next step was to consider the boundaries of the test domains, and any further test cases necessary because it is recognised that problems frequently arise at the boundaries of the test domains. The technique described above can be shown to be able to generate test cases which are *reliable* and *valid*. However, the main problem with this technique is that it is not capable of being used for generating test cases automatically.

Zin et al's[16] approach for generating test cases from a formal specification is different from the one described by Hall. Instead of generating test cases directly from the specification, they translated the specification into a Prolog implementation, and then used the Prolog implementation to generate the test cases. The advantage of using this approach is that the test case generation can be done almost automatically.

By combining the classification tree method with the disjunctive normal form approach, Singh et al[17] show that they are able to develop a tool to aid the generation of test cases from Z specifications.

In general, there are several issues that need to be considered when we plan to generate test cases. In this section, we discuss some of the issues.

Oracle Problem : This is a problem regarding the verification of the correctness of the outputs obtained during the testing activity. Only a few testing

techniques address this problem.[18]

Overhead : Weyuker[19] said that "exhaustive testing is typically prohibitively expensive, because a formula of n variables would require 2^n distinct test cases." Madrioli et al[18] (when considering this huge number of test cases) said that it, "makes exhaustive testing impossible in most practical cases, even for finite interpretation domains." Foster[20] said that small sets of test case should be chosen "provided test cases are selected that require each variable value to individually affect the result." Smaller test sets that interest Elaine et al[19] are, "smaller than exhaustive test sets, but would nonetheless be highly effective at detecting faults".

Effectiveness : Jalote[21] said that "the effectiveness of any testing process is highly dependent on the choice of test case." If proper test cases are not provided, the test goal may not be achieved. Mutation analysis is used by Elaine et al[19] in their test case evaluation. Mutant analysis was first introduce by DeMillo et al.[22] Elaine et al described the analysis as it "starts with a program to be tested and makes numerous small changes to it, creating a set of mutant programs. Each mutant is then run on a test set that is being evaluated to see whether the test data are comprehensive enough to distinguish the original program from each inequivalent mutant."

Automation : If we can automate the generation of the test cases, we can generate the whole process of testing automatically. Manual testing methods are proving to be ineffective in today's software environment.[23] It has been shown that software quality can not be effectively achieved using manual testing techniques.

Significance : Additionally, in educational issues, we are considering the progress of the student. It is not practical to evaluate the student solution by using yes or no answers. We should grade the level of their correctness. Therefore we should consider the significance of the chosen test case.

By having a model solution as a guide, we might use the above ideas and issues to direct our research in producing the test case automatically.

7.4.5. Technique for allocating marks

Another interesting research area which could benefit from the process of automatic assessment is the allocation of weights for every test case. We might decide to allocate different marks for boundary type test cases compared to non-boundary test cases. However, the extent that the mark is different between both test cases is an interesting issue to discuss. This section discusses the rationale of choosing the right weight for the analysis.

Before going further, there are some issues that need to be highlighted:

How many test cases are needed.

What criteria to select the test cases.

How to distribute the marks.

What basis do we use to distribute the marks

The first two questions have been discussed in the previous section. In this section we would like to show how the subjectivity of distribution of marks can be reduced and rationalised. The steps can be summarised as follows:

1 Obtain a set of test cases.

2 Categorise the test cases according to their significance.

3 Allocate symbolic marks (represent by variables) for each category.

4 Define the marks pattern which is in equation form.

5 Derive the absolute value for the symbolic marks using a Linear Programming technique.

This idea will be illustrated by using the following example.

```
Write a Z schema which accept any number bigger than 5
```

```
and smaller than 10.
```

The answer might be

Test

$n : \mathbb{Z}$

$n > 5 \land n < 10$

As regards the testing technique, we will give several inputs and observe the behaviour of the schema.

Step 1: Obtaining test cases

For non-boundary cases

[td1]-if the input falls in the range $5<n<10$, the schema should yield value *true*.

[td2]-if the input falls in the range $n<5$, the schema should yield value *false*.

[td3]-if the input falls in the range $n>10$, the schema should yield value *false*.

and for boundary cases

[td4]-if the input is equal to the first border line, $n=5$, the schema should yield value *false*.

[td5]-if the input is equal to the second border line, $n=10$, the schema should yield value *false*.

Step 2: Categorising the test cases

We decide that

- all non-boundary test cases are equally significant.
- all boundary test cases are equally significant.
- non-boundary test cases are significantly different from boundary test cases.

Step 3: Allocating symbolic marks

From the above arrangement (step 2), we only need to have two different marks, which we choose to be A and B. Variable A will represent the mark awarded if the testing satisfies non-boundary test cases, and variable B will represent the mark awarded if the testing satisfies boundary test cases. In other words:

if the testing satisfies [td1], [td2] or [td3], award A marks for each of them.

if the testing satisfies [td4] or [td5], award B marks for each of them.

Step 4: Defining a pattern

Clearly if the student answer fulfills the question criteria (i.e their answer satisfies all the test cases) they will be awarded 100%, i.e

$$3A+2B=100$$

If the student satisfies half of the answers, we intend to give around 50%. For example, the student answer is $n>5$. Therefore it satisfies [td1], [td2] and [td4].

$$2A+B\approx 50$$

We intend to give 70% to the student whose answer only fails at the boundaries. For example, if the student answer is $n\geq 5 \wedge n\leq 10$. It satisfies all the test cases except for boundary, i.e [td4] and [td5].

$$3A\approx 70\%$$

If the student fulfills half of the answer without the boundaries, we intend to give around 40%. For example, the student answer is $n\geq 5$. It satisfies [td1] and [td2].

$$2A\approx 40$$

And if the student only fulfills the boundaries, we give 20%. For example, the student answer is $n\neq 5 \wedge n\neq 10$. This only satisfies [td4] and [td5].

$$2B\approx 20$$

In summary, we get

[1] $3A+2B=100\%$

[2] $2A+B \approx 50\%$

[3] $3A \approx 70\%$

[4] $2A \approx 40\%$

[5] $2B \approx 20\%$

Step 5: Deriving the value

By using trial and error, we can derive values for A and B that tolerably satisfy the intention.

First trial: from [1] and [5], we get

A = 26.67% and B = 10%

By replacing the value of A and B in other equations, we get

from [2], $2A+B=63.33\%$ (+13.33% deviation from the original value i.e 50%)

from [3], $3A=80\%$ (+10% deviation from 70%)

from [4], $2A=53.33\%$ (+13.33% deviation from 40%)

Second trial: from [1] and [4], we get

A = 20% and B = 20%

By replacing the value of A and B in other equations, we get

from [2], $2A+B=60\%$ (+10% deviation from 50%)

from [3], $3A=60\%$ (-10% deviation from 70%)

from [5], $2B=40\%$ (-20% deviation from 20%)

We might be able to continue this trial by using other values. However, the best value (which is close to our earlier intention) can be achieved by using Linear Programming. This can be explained as follows.

We have

$3A+2B=100$

with the conditions as below:

$2A+B=50+r$

$3A=70+s$

$2A=40+t$

$2B=20+u$

where r, s, t and u are errors. In the above case r, s, t and u can take positive or negative value. Therefore,

$r=2A+B-50$

$s=3A-70$

$t=2A-40$

$u=2B-20$

The value of the errors (r, s, t and u) will be controlled so that it can be in the range of n% from the intended value. If n=20

r: $-n\%(50)<r<n\%(50)$

$-10<r<10$

s: $-n\%(70)<s<n\%(70)$

$-14<s<14$

t: $-n\%(40)<t<n\%(40)$

$-8<t<8$

u: $-n\%(20)<u<n\%(20)$

$-4<u<4$

Now we can use Linear Programming to solve the problem:

$r=2A+B-50$ then

$$-10<2A+B-50<10 \quad [6]$$

$s=3A-70$ then

$$-14<3A-70<14 \quad [7]$$

$t=2A-40$ then

$$-8<2A-40<8 \quad [8]$$

$u=2B-20$ then

$$-4<2B-20<4 \quad [9]$$

From [6], [7], [8] and [9], we can conclude that A and B fall in the range

$$8<B<12$$

$$18.67<A<24$$

However, none of the values in range A and B will satisfy $3A+2B=100$. Therefore, we take the value A and B which is very near to that range and satisfy the main equation $3A+2B=100$. We choose to set $B=12$ and $A=25.33$. Finally we can claim that value A and B that we choose give errors of around 20% from the intended value.

In this section, we have presented a formal way to decide what marks we should give. However human instinct is still being used in this technique. This idea has not yet been proved to be effective, nonetheless it seems useful especially in supporting a general automatic marking system.

7.5. Final remarks

In this thesis, we have demonstrated the feasibility of assessing Z specifications automatically. A system has been developed as an integral part of this research project. The existing type and syntax checker and editing tool as well as the marking system to support the on-line learning are shown to be beneficial to students. The marking system has been tested experimentally against real student answers. It successfully reveals that its behaviour in awarding marks is similar to human marking.

References

1. A.M. Zin, *ZFDSS: A Formal Development Support System based on the Liberal Approach*, 1994. PhD Thesis, University of Nottingham, UK
2. L. Sterling, P. Ciancarini, and T. Turnidge, "On the Animation of "Not Executable" Specifications by Prolog," *International Journal of Software Engineering and Knowledge Engineering*, vol. 6, no. 1, pp. 63-87, 1996.
3. R. Knott and P. Krause, "The implementation of Z specifications using program transformation systems: The SuZan project," in *The Unified Computation Laboratory* , ed. C. Rattray & R. Clark, vol. 35, pp. 207-220, IMA Conference Series, Clarendon Press, Oxford, UK, 1992.
4. V. Doma and R. Nicholl, *EZ : A system for automatic prototyping of Z specifications,* 551, pp. 189-203, Lecture Notes in Computer Science, Noordwijkerhout, Springer-Verlag, Berlin, October 1991.
5. A. Diller, *Z An Introduction to Formal Methods,* John Wiley & Sons, 1994.
6. M. Johnson and P. Sanders, "From Z specifications to functional implementations," *Proc. 4th Z Users Meeting*, pp. 86-112, Springer-Verlag, 1989.
7. P. Henderson, "Functional programming, formal specification, and rapid prototyping," *IEEE Transaction*, vol. 12, no. 2, pp. 241-249, 1986.
8. A.M. Gravell and P. Henderson, "Why execute formal specifications?," *Proc. Mathematical Structures for Software Engineering*, pp. 165-184, Oxford University press, 1991.
9. A. Gravell and P. Henderson, "Executing formal specifications need not be harmful," *Software Engineering Journal*, vol. 11, no. 2, pp. 104-110, March 1996.
10. N.E. Fuchs, "Specifications are (preferably) executable," *Software Engineering Journal*, vol. 7, no. 5, pp. 323-334, September 1992.

11. I.J. Hayes and C.B. Jones, *Specifications are not (necessarily) executable &J Software Engineering Journal*, 4, pp. 330-338, November 1989.

12. E. Foxley, "The Ceilidh spelling checker," *LTR Report*, Computer Science Dept, Nottingham University.

13. L.C. Paulson and with contribution bu Tobias Nipkow, *Isabelle: A Generic Theorem Prover,* Springer Lecture Notes in Computer Science 828..

14. M. Saaltink, "Effective Use of Z/EVES," *Tutorial Programme of Z User Meeting '98*, Berlin, Germany, September 1998.

15. P.A.V. Hall, "Towards Testing with Respect to Formal Specifications," *Proc. of Second IEE/BCS Conference: Software Engineering 88*, pp. 159-163, London, 1988.

16. A.M. Zin, A. Al-Amayreh, and E. Foxley, "An Approach to Specification-based Testing Systems," *Proceedings of Software Quality Engineering Conference*, Udine, Italy, 6 May 1997.

17. H. Singh, M. Conrad, and G. Egger, "Tool-Supported Test Case Design Based on Z and the classification-Tree Method," *Workshop "Tool Support for System Development and Verification"*, Bremen, 1996.

18. D. Mandrioli, S. Morasca, and A. Morzenti, "Generating Test Cases for Real-Time Systems from Logic Specifications," *ACM Transactions on Computer Systems*, vol. 13, no. 4, pp. 365-398, Plotecnico di Milano, November 1995.

19. E. Weyuker, T. Goradia, and A. Singh, "Automatically Generating Test Data from a Boolean Specification," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 353-363, May 1994.

20. K.A. Foster, "Sensitive test data for logic expressions," *ACM SIGSOFT Software Engineering Notes*, vol. 9, no. 2, pp. 120-126, April 1984.

21. P. Jalote, "Testing the Completeness of Specification," *IEEE Transactions on Software Engineering*, vol. 15, no. 5, pp. 526-531, May 1989.

22. R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on test data selection: Help for practiting programmer," *Computer*, April 1978.

23. E. Sabbatini, M. Crubellati, and S. Siciliano, "Automating test by adding formal specification: An experience for database bound applications," in *Software Quality Engineering*, ed. C. Tasso, R.A. Adey and M. Pighin, Udine, Italy, May 1997.

Bibliography

Austin, S. and Parkin, G.I., Formal Methods: A Survey, Report, The National Physical Lab., Middlesex (31 March 1993).

Balzer, R. and Goldman, N., Principles of Good Software Specification and their Implications for Specification Languages, *Proceedings of IEEE Conference on Specifications of Reliable Software*, pp. 58-67 (1979).

Barden, R., Stepney, S., and Cooper, D., *Z in Practice*, Prentice Hall (1994).

Basili, V.R., Selby, R.W.Jr., and T., Philips, Metric analysis and validation across Fortran projects, *IEEE transaction Software Engineering SE-9*, pp. 652-663 (Nov 1983).

Basili, V.R., Tutorial on Models and Methods for Software Management and Engineering. , *IEEE Computer Society Press*, (1980).

Beizer, B., *Software system testing and quality assurance*, Van Nostrand Reihold (1984).

Benford, Steve, Burke, Edmund, and Foxley, Eric, *Courseware to support the teaching of programming*, TLTP Conference, University of Kent at Canterbury 1992.

Benford, S., Burke, E., Foxley, E., Gutteridge, N., and Zin, A.M., Ceilidh as a Course Management Support System, *Journal of Educational Technology Systems* **22**(3)(September 1993).

Benford, S., Burke, E., Foxley, E., Gutteridge, N., and Zin, A.M., CEILIDIH: A Course Administration and Marking System, *Proceedings of International Conference in Computer based Learning in Science*, (1993).

Benford, S., Burke, E., Foxley, E., Gutteridge, N., and Zin, A.M., Early experiences of computer aided assessment and administration when teaching computer

programming, *Association for Learning Technology Journal* 1(2) pp. 55-70 (1993).

Bottaci, L. and Jones, J., *Formal Specification Using Z: A Modelling Approach,* International Thomson Publishing (1995).

Brusilovsky, P. and Pesin, L., ISIS-Tutor: An adaptive hypertext learning environment, *Proc. JCKBSE'94, Japanese-CIS Symposium on knowledge-based software engineering.* , pp. 83-87 (May 10-13, 1994).

Burgess, R.S., *An Introduction to Program design using JSP,* Hutchinson & Co. Publisher Ltd (1984).

Burke, E. and Foxley, E., *Logic and its Applications,* Prentice Hall Europe (1996).

Coleman, M. and Pratt, S., *Software Engineering for Students 1986,* Chartwell-Bratt Ltd. (1986).

Cooke, D., Gates, A., Demirors, E., Demirors, O., Tanik, M.M., and Kramer, B., Languages for the Specification of Software, *Journal of System Software* **32**(3) pp. 269-308 (1996).

Curtis, W., Management and Experimentation in Software Engineering, *Proceeding of the IEEE* **68**(9)(September 1980).

Dean, C.N. and Hinchey, M.G., Introducing Formal Methods Through Role Play, *ACM SIGCSE Bulletin* **27**(1) pp. 302-306 (March 1995).

DeMillo, R.A., Lipton, R.J., and Sayward, F.G., Hints on test data selection: Help for practiting programmer, *Computer*, (April 1978).

Dijkstra, E.W., Foreword, in *Teaching and Learning Formal Methods*, ed. C.N. Dean & M.G. Hinchey,, San Diego, California, US (1996).

Diller, A., *Z An Introduction to Formal Methods,* John Wiley & Sons (1994).

Doma, V. and Nicholl, R., *EZ : A system for automatic prototyping of Z*

specifications, Lecture Notes in Computer Science, Noordwijkerhout, Springer-Verlag, Berlin (October 1991).

, Educational Issues relating to Formal Methods, *Educational Issues Session of Z Users Meetings '94*, ().

Faidhi, J.A.W., The complexity analysis of Pascal programs and the application to a university teaching environment, *PhD Thesis*, University of Brunel, (1986).

Fields, B. and Elovang-Goransson, M., A VDM Case Study in mural, *IEEE Trans. Software Eng.* **18**(4) pp. 279-295 (Apr. 1992).

Finney, K., Mathematical Notation in Formal Specification: Too Difficult for the Masses?, *IEEE Transactions on Software Engineering* **22**(2) pp. 158-159 (February 1996).

Foster, K.A., Sensitive test data for logic expressions, *ACM SIGSOFT Software Engineering Notes* **9**(2) pp. 120-126 (April 1984).

Foubister, S.P., Michaelson, G.J., and Tomes, N., Automatic assessment of elementary Standard ML programs using Ceilidh, *Journal of Computer Assisted Learning* **13** pp. 99-108 (1997).

Foxley, E., Burke, E., Higgins, C., and Gibbon, C., The Ceilidh System:A General Overview as at December 1996, *LTR Report*, (1996).

Foxley, E., The Ceilidh spelling checker, *LTR Report*, ().

Foxley, E., Salman, O., and Shukur, Z., The Automatic Assessment of Z Specification, *Proceedings of ITiCSE '97 Conference*, (June 1997).

Foxley, E. and Zin, A.M., *Zpp - A Troff Preprocessor for Typesetting Z Specifications*, Nottingham University Computer Science 1990.

Franklin, M.E., Automatic analysis of the structure of Pascal programs written by

novices, *MPhil Thesis*, University of Sheffield, (1987).

Fuchs, N.E., Specifications are (preferably) executable, *Software Engineering Journal* 7(5) pp. 323-334 (September 1992).

Garlan, D., Making formal methods education effective for professional software engineers, *Information and Software Technology* **37**(5-6) pp. 261-268 (1995).

Goguen, J.A., Parameterized Programming, *IEEE Transactions on Software Engineering* **10**(5) pp. 528-543 (1984).

Gravell, A. and Henderson, P., Executing formal specifications need not be harmful, *Software Engineering Journal* **11**(2) pp. 104-110 (March 1996).

Gravell, A.M., What is a Good Formal Specification?, *Fifth Annual Z User Meeting*, (17 December 1990).

Gravell, A.M. and Henderson, P., Why execute formal specifications?, *Proc. Mathematical Structures for Software Engineering*, pp. 165-184 Oxford University press, (1991).

Haigh, N.P.H., Providing tool support for Z, *Software Tools: Improving Applications*, pp. 185-191 (June 1987).

Hall, P.A.V., Towards Testing with Respect to Formal Specifications, *Proc. of Second IEE/BCS Conference: Software Engineering 88*, pp. 159-163 (1988).

Halstead, M., *Elements of Software Science*, Elsevier Scientific Publishing Co. (1977).

Hayes, I.J. and Jones, C.B., *Specifications are not (necessarily) executable &J Software Engineering Journal.* November 1989.

Henderson, P., Functional programming, formal specification, and rapid prototyping, *IEEE Transaction* **12**(2) pp. 241-249 (1986).

Henry, S. and Kafura, D., On the relationship among three software metrics, *Perform. Eval. Rev.* , (10, 1) pp. 81-88 (Spring 1981).

Hung, S., Kwok, L., and Chung, A., New Metrics for Automated Programming Assessment, *IFIP Transactions A-Computer Science and Technology* **40** pp. 233-243 (1993).

Jalote, P., Testing the Completeness of Specification, *IEEE Transactions on Software Engineering* **15**(5) pp. 526-531 (May 1989).

Johnson, M. and Sanders, P., From Z specifications to functional implementations, *Proc. 4th Z Users Meeting*, pp. 86-112 Springer-Verlag, (1989).

Jones, C., A Survey of Programming Design and Specification Techniques, *Proceedings of IEEE Conference on Specification of Reliable Software*, pp. 91-103 (1979).

Kearney, J.K., Sedlmeyer, R.L., Thompson, W.B., and Gray, M.A., Software Complexity Measurement, *Communications of the ACM* **29**(11) pp. 1044-1050 (September 1986).

Kemmerer, R.A., Testing Formal Specifications to Detect Design Errors, *IEEE Transactions on Software Engineering* **11**(1) pp. 32-43 (January 1985).

Kernighan, B.W. and Cherry, L.L., A System for Typesetting Mathematics, *Communications of the ACM* **18** pp. 151-157 (1975).

King, J.C., Symbolic execution and Program testing, *Communication of the ACM* **19** pp. 385 - 394 (July 1976).

Knott, R. and Krause, P., The implementation of Z specifications using program transformation systems: The SuZan project, pp. 207-220 in *The Unified Computation Laboratory* , ed. C. Rattray & R. Clark,IMA Conference Series, Clarendon Press, Oxford, UK (1992).

Leite, J.C.S. do Prado and Freeman, P.A., Requirements Validation Through Viewpoint Resolution, *IEEE Transactions on Software Engineering* **17**(12) pp. 1253-1269 (December 1991).

Lesgold, A.M., Lajoie, S.P., Bunzo, M., and Eggan, G., SHERLOCK: A coached practice environment for an electronics troubleshooting job, pp. 201-238 in *Computer assisted instruction and intelligent tutoring systems: Shared issues and complementary approaches* , ed. J. Larkin & R. Chabay ,Lawrence Erlbaum Associates, Hillsdale, NJ (1992).

Looi, C.K., Automatic program analysis in a Prolog intelligent teaching system, *PhD Thesis*, University of Edinburgh, (May 1988).

Macdonald, R., Z Usage and Abusage, *Report 91003 Royal Signals and Radar Establishment*, (February 91).

Mandrioli, D., Morasca, S., and Morzenti, A., Generating Test Cases for Real-Time Systems from Logic Specifications, *ACM Transactions on Computer Systems* **13**(4) pp. 365-398 (November 1995).

McCabe, T.J., A Complexity Measure, *IEEE Transactions on Software Engineering* **2**(4) pp. 308-320 (December 1976).

McCall, J., Richards, P., and Walters, G., *Factors in Software Quality, 3 Vols.*, NTIS AD-A049-014,015,055 1977.

Michaelson, G., Automatic analysis of functional program style, *Australian Software Engineering Conference* **13** pp. 38-46 (1996).

Myers, G., *The Art of Software Testing*, Wiley ().

Naur, P. and Randell, B., *Software Engineering*, NATO 1968.

Oman, P.W. and Cook, C.R., A Paradigm for Programming Style Research, *SIGPLAN Notices* **23**(12) pp. 69-79 ().

Parker, C., Z Tools Catalogue, ZIP/BAe/90/020, Software Technology Dept, British Aerospace (10 May 1991).

Paulson, L.C. and Nipkow, with contribution bu Tobias, *Isabelle: A Generic Theorem Prover*, Springer Lecture Notes in Computer Science 828. ().

Pressman, R.S., *Software Engineering*, McGraw-Hill Company Europe (1992).

Rann, D., Turner, J., and Whitworth, J., *Z: A Beginner's Guide*, Chapman & Hall (1994).

Ratcliff, B., *Introduction Specification Using Z : A Pracatical Case Study Approach*, McGraw Hill International (1994).

Redish, K.A. and Smyth, W.F., Program Style Analysis: A Natural By-Product of Program Compilation, *Communications of the ACM* **29**(2) pp. 126-133 (February 1986).

Redish, K.A. and Smyth, W.F., Evaluating Measures of Program Quality, *The Computer Journal* **30**(3)(1987).

Rees, M.J., Automatic Assessment Aid for Pascal Programs, *SIGPLAN Notices* **17** (10) pp. 33-42 (October 1982).

Richardson, D.J., Aha, S. Leif, and O'Malley, T.O., Specification-based Test Oracles for Reactive Systems, *Proc. of the 14th ICSE International Conference on Software Engineering* , pp. 105-118 IEEE/ACM, (1992).

Saaltink, M., Effective Use of Z/EVES, *Tutorial Programme of Z User Meeting '98*, (September 1998).

Sabbatini, E., Crubellati, M., and Siciliano, S., Automating test by adding formal specification: An experience for database bound applications, in *Software Quality Engineering*, ed. C. Tasso, R.A. Adey and M. Pighin,, Udine, Italy (May 1997).

Sadeghipour, S., Test Case Generation on the basis of Formal Specifications, *Proc. FEmSys '97*, (1997).

Schneidewind, N.F., Standards, *Computer*, (April 1993).

Schwarz, E., Brusilovsky, P., and , G. Weber, World-wide intelligent textbooks. , *Proceedings of ED-TELEKOM 96 - World Conference on Educational Telecommunications*, pp. 302-307 Charlottesville, VA: AACE., (1996).

Shneiderman, B., *Software Psychology: Human Factors in Computer and Information Systems,* Winthrop Publishers (1980).

Shukur, Z., Burke, E., and Foxley, E., Applying Z Specification Coursework On-Line, *Proceedings of AltC-98 Conference*, (September 1998).

Shukur, Z., Burke, E., and Foxley, E., The Automatic Assessment of Formal Specification Coursework, *Journal of Computing in Higher Education*, ((will be published in) August 1999).

Shukur, Z., Burke, E., and Foxley, E., Automatic Marking System for Z Specifications, *Proceedings of PROGRESS 98 Conference*, (March 1998).

Shukur, Z., Burke, E., and Foxley, E., Inspecting the correctness of specification through system-state analysis, *IEEE transaction on Software Engineering*, ((will be submitted) 1999).

Shukur, Z., Burke, E., and Foxley, E., Managing Z Specification Coursework On-line, *Journal of Computers in Mathematics and Science Teaching*, ((submitted) 1999).

Singh, H., Conrad, M., and Egger, G., Tool-Supported Test Case Design Based on Z and the classification-Tree Method, *Workshop "Tool Support for System Development and Verification"*, (1996).

Spivey, J.M., *The Z notation : Reference Manual,* Prentice Hall (1988).

Spivey, J.M., *Understanding Z: A Specification language and its formal semantics,* Cambridge University Press (1988).

Spivey, J. M., *The Z Notation : A Reference Manual,* Prentice-Hall, Inc. 1989.

Steggles, P. and Hulance, J., *Z Tools Survey,* Imperial Software Technology Ltd & Formal System (Europe) Ltd (June 1994).

Sterling, L., Ciancarini, P., and Turnidge, T., Specifications by Prolog"" On the Animation of "Not Executable" Specifications by Prolog, *International Journal of Software Engineering and Knowledge Engineering* **6**(1) pp. 63-87 (1996).

Valentine, S.H., The programming language Z--, *Information and Software Technology* **37**(5-6) pp. 293-301 (1995).

Verth, P.B. Van, A System for Automatically Grading Program Quality, *SUNY (Buffalo) Technical Report,* (1985).

Weyuker, E., Goradia, T., and Singh, A., Automatically Generating Test Data from a Boolean Specification, *IEEE Transactions on Software Engineering* **20**(5) pp. 353-363 (May 1994).

Whitty, R., Structural Metrics for Z Specifications, *Fourth Annual ZUM,* (15 December 1989).

Woolf, B., Intelligent tutoring systems: A Survey, pp. 1-41 in *Exploring artificial intelligence: Survey talks from the National Conferences on artificial intelligence,* ed. H. E. Shrode,Morgan Kaufmann Publishers Inc. , San Mateo, CA (1988).

Wordsworth, J.B., Education in formal methods for software engineering, *Information and Software Technology* **29**(Jan-Feb 1987).

Wordsworth, J.B., An industrial perspective on educational issues relating to formal methods, pp. 1-9 in *Teaching and Learning Formal Methods,* ed. C.N. Dean & M.G. Hinchey,, San Diego, California, US (1996).

Yau, S.S. and Collofello, J.S., Some Stability Measures for Software Maintenance, *IEEE Transactions on Software Engineering* **6**(6) pp. 545-552 (November 1980).

Youngblut, C., Government-Sponsored Research and Development Efforts in the Area of Intelligent Tutoring Systems, (IDA Paper P-3003), Institute for Defense Analyses, Alexandria, VA. (September, 1994).

Zin, A.M., Al-Amayreh, A., and Foxley, E., An Approach to Specification-based Testing Systems, *Proceedings of Software Quality Engineering Conference*, (6 May 1997).

Zin, A.M. and Foxley, E., Software Tools for Animating a Z Specification, *Sains Malaysiana* **24**(4) pp. 67-89 (1995).

Zin, A.M. and Foxley, E., Analyse - An automatic program assessment system, *Malaysian Journal of Computer Science* **7** p. 123 (1994).

Zin, A. M. and Foxley, E., Automatic Program Quality Assessment System, *Proceedings of the IFIP Conference on Software Quality*, (March 1991). S P University, Vidyanagar, INDIA

Zin, A. M. and Foxley, E., The Oracle Program, *LTR Report*, (1992).

Zin, A.M., *ZFDSS: A Formal Development Support System based on the Liberal Approach*, PhD Thesis, University of Nottingham, UK 1994.

, ZUM '98: The Z Formal Specification Notation, *Proceedings of 11th International Conference of Z Users*, (1493)Springer, (September 1998).

Appendix A
The Syntax Definition

In this definition, all terminal symbols are placed in double qoutes, for example "NL". A symbol or symbols appear in [] are considered to be optional, and {} denotes that a symbol or symbols can appear zero or more times.

Para	::=	Schema_Name "NL" Mark_Grammar
Mark_Grammar	::=	Number "COLON" Predicate-0 "NL" Mark_Grammar
	|	NIL

PREDICATE

Predicate-0	::=	"TRUE"
	|	"TRUE" "COLON" "STRING"
	|	"TRUE" Predicate-2
	|	"FALSE"
	|	"FALSE" "COLON" "STRING"
	|	"FALSE" Predicate-2
Predicate-1	::=	Pre-Rel Expression Predicate-2
	|	"TRUE" Predicate-2
	|	"FALSE" Predicate-2
	|	"NOT" Predicate-1 Predicate-2
	|	"(" Predicate ")" Predicate-2
	|	Expression { In-Rel Expression } Predicate-2
Predicate-2	::=	"AND" Predicate-1 Predicate-2
	|	"OR" Predicate-1 Predicate-2
	|	"IMP" Predicate-1 Predicate-2

	| "EQV" Predicate-1 Predicate-2
	| NIL

EXPRESSION

Expression	::=	Expression-1 { "CROSS" Expression-1 } Exp
Exp	::=	In-Gen Expression Exp
	|	NIL
Expression-1	::=	"POWER" Expression-3 Exp-1
	|	Pre-Fun Expression Exp-1
	|	Pre-Gen Expression-3 Exp-1
	|	"-" Expression-3 Exp-1
	|	Expression-3 Post-Fun Exp-1
	|	Expression-3 "ITERATE" Expression Exp-1
	|	Expression-3 Exp-2 Exp-1
	|	Expression-3 Exp-1
Exp-1	::=	In-Fun Expression-1 Exp-1
	|	NIL
Exp-2	::=	Expression-3 Exp-2
	|	NIL
Expression-3	::=	"(" Expression ")"
	|	Expression-4
Expression-4	::=	Ident-0
	|	"EMPTYSET"

```
                |   "SET" Expression-4 { "," Expression-4 } "TES
                |   "EMPTYSEQ"
                |   "SEQ" Expression-4 { "," Expression-4 } "QES
                |   "EMPTYBAG"
                |   "BAG" Expression-4 { "," Expression-4 } "GAB"
                |   "TUP" Expression-4 { "," Expression-4 } "PUT"

Ident-0         ::= Word { Decoration }
                |   Number
                |   "[" Ident-1 "]"

Ident-1         ::= Ident-0 Ident-2
                |   NIL

Ident-2         ::= "," Ident-1
                |   NIL

Decoration      ::= "PRIME" | "!" | "?"
```

Appendix B
Z Coursework Questionaire

UserId (optional):

Mark the answer (with '#') which best describe you

for example:

0. Are you a student?

A. Yes #

B. No

FSP-Ceilidh Questionaire

EDITOR

---

1. Have you ever used a roff preprocessor before?

A. Yes

B. No #

2. How easy did you find it to write a Z specification using the roff format AT FIRST?

A. Easy

B. Fair #

C. Hard

D. Very difficult

3. How easy did you find it to write a Z specification using the roff format NOW?

A. Easy

B. Fair #

C. Hard

D. Very difficult

4. How easy did you find it to write a Z specification by hand (using pen & pencil)?

A. Easy #

B. Fair

C. Hard

D. Very difficult

5. Do you think

A. It is worth the effort to write in the roff format or #

B. would you prefer to submit handwritten works

6. Are you satisfied with the response time of Ceilidh when you amend your specification, refresh it and redisplay in Ghostview?

A. Yes

B. No #

7. Are you familiar with the WORDS (e.g Microsoft WORDS) mathematical symbol library?

A. Yes #

B. No

8. If there existed a Z editor would you

A. prefer to use it #

B. prefer the roff format

C. do not know (because you have never used WORDS mathematical library)

TYPE and SYNTAX CHECKER (tc)

---

9. Did you ever USE tc when doing your coursework?

A. Yes #

B. Never

10. Was tc helpful in solving your coursework?

A. Only for some exercises

B. For all exercises

C. It can help in some problems such as detecting any undeclared variable. However, sometimes it produce error messages that I could not understand #

11. In general, do you think you need tc?

A. Need tc #

B. Do not need it

12. If there existed an IDEAL tc, do you think you would use it?

A. Yes #

B. No

ANIMATION

Animation is when the specification would be 'executed'.

You can give an input to the schema and observe the output.

13. If there existed an animator (tools that can make the specification executable), do you think you would use it?

A. Yes #

B. No

C. Do not know

GENERAL

---

14. If you were to specify a system, would you use

A. Z Specification #

B. Natural language

C. SSADM

D. Other specification

15. If Z is not your choice, can you give the reason

16. Do you have any other comments about FSP-Ceilidh?

none.

Appendix C
Student Feedback for the Questionaire

Below is the result for **Question 1 to 14.**

Question	Student Id									
	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10
1	B	A	B	A	A	B	B	B	B	A
2	C	C	B	B	B	B	C	B	C	C
3	B	B	B	A	B	A	B	B	B	B
4	A	A	A	B	E	A	B	A	B	B
5	A	B	A	A	A	A	A	A	A	A
6	A	A	B	B	A	B	A	B	A	B
7	A	A	B	B	B	B	B	A	A	B
8	A	A	C	C	C	C	C	A	A	A
9	A	-	A	A	A	A	A	A	A	A
10	A	-	C	C	A	C	C	C	C	A/C
11	B	-	B	A	A	A	A	A	A	A
12	A	A	A	A	A	A	A	A	A	A
13	A	A	B	A	A	A	A	A	A	A
14	D	B/C	A/D	-	B	B	A	A	A/B/D	B

The following is feedbacks from the students regarding to question 15 and 16. The sentences in the quote is the original comments.

Question 15 : Reason of why Z is not my choice

- "Takes many hours to specify a resonably large system, because you either do it 100% accurately or not at all in Z."
- "need more experience in writing Z specification."
- "It's only useful if a) I understood it well enough to be completely confident & b) if everyone I wanted to communicate with do too."
- "In most situations giving a Z specification for a system would be too tedious. Unless there are some high risks involved in using the final system I don't think it can be justified."
- "Other specification methods, like SSADM (and natural language) provide an 'at a glance' image of the system via flow charts or E-R diagrams etc. I feel that Z does not offer this, as many equations need to be decyphered before understanding the system, and it is hard to see the complete picture all at once with Z. A better idea would be to use Z alongside other methods. For instance, within SSADM, Z could be used for certain modules where a strict mathematical specification is required."
- "I have nothing against Z but personally I feel that natural language is easier to understand and things are easier to express using natural language. I don't mind using Z to specify easy systems."
- "nice but not really powerful compared to UNITY; it looks like a database specification language, there is nothing about concurrency programming, real time and shared ressources."

Question 16 : Comments about FSP-Ceilidh

- "An exam would have been nice. tc should be improved. Ceilidh is too slow, and the text interface is outdated."
- "provide a menu/list of all available commands

  e.g .ZS - start schema

.ZE - end schema

GUNION -

CSET - {

TESC - }"

- "TC should be better. It will be very helpful provided it works fine."
- "No, I'd rather forget the whole sordid ordeal thank you :>"
- "Some guidance on how to format the text (e.g. for adding comments and titles) we used in our documents using roff macros would have been useful. Sometimes the documentation for the preprocessor wasn't entirely accurate, for example 'prime' instead of 'PRIME' and the undocumented 'SUCHTHAT' to produce a |. But on the whole I think the system was good. It would have been nice to be able to write stand-alone predicates in the documents, i.e. outside Z schemas. When I tried this using .ZF or .ZC the preproceesor crashed. Occasionaly I think better thought should have gone into the exercises being set. Especially in the case of Ex6.4 from the book, which was set twice as the basis to a coursework exercise. Just before the 2nd handin we were told to just say why the exercise was trival and no changes where needed - This effectively made the 1st exercise using this question non-existant!!"
- "I think if the tc could be improved then it would be a great help for doing the coursework. Since it is not working properly, I think it should not be offered for the time being."

Appendix D
Compilation of Z Exercises

The following are four case studies that were given to the students. The details of the case studies can be found in the respective references.

- Case study 1: Class Homework Problem[1]
- Case study 2: A Video-rental Shop[2]
- Case study 3: A Car Ferry Terminal[2]
- Case study 4: Height and Weight Problem[2]

The exercises are grouped into three: the accepted exercises, the accepted exercises with some editing and the rejected exercises. Regarding to the rejected exercises, the reason of this situation will be explained.

Accepted exercises

Exercise 1: Case study 1

Amend the schema *ClassHomework* to limit the number of students in the class to be less than or equal to 20.

Exercise 2: Case study 1

Write two possible state schemas for a similar system of students with work either handed in or not handed in, in which the class consists of some special honours students and some joint honours students.

Exercise 3: Case study 1

Write a schema for the operation *add_student* defined as follows: There shall be a command to add a new student to the class. The name of the student will be the input to the command. It is to be assumed that the new student has not handed in any work. The new student must not already be a member of the class.

Exercise 4: Case study 1

Write a schema for the operation *add_student* defined as follows: There shall be a command to add a new student to the class. The name of the student will be input to

the command, together with a Boolean which will take the value *T* if work is to be handed in at the time of registration. The new student must not already be a member of the class.

Exercise 5: Case study 1

Write a schema to list the students who have handed in their coursework.

Accepted exercises with some editing

Exercise 6: Case study 2

Write a schema for the command collect_reserved defined as follows: Allow a user to collect a video he/she has reserved. Check details and adjust records. Make sure the video is not on loan.

Exercise 7: Case study 2

Write a schema for the command return_video defined as follows: Customer returns video. Calculate and output charge and inform user whether this video is reserved.

Exercise 8: Case study 2

Write a schema for the command add_video defined as follows:. Allocate a unique reference for a new video and update the records. Output the new video reference.

Exercise 9: Case study 3

Write a schema for the command list_ferry_spaces defined as follows: Given a ferry reference, give the number of spaces on it. Report an error if the ferry reference is invalid.

Exercise 10: Case study 3

Write schema for the command can_ferries_combine defined as follows: The inputs are two ferry references. Give the message 'OK' if there is enough space on the second ferry for the vehicles currently scheduled for the first ferry, and the message 'Not_enough_space' otherwise. Report an error if either of these ferry references is invalid.

Rejected exercises

Exercise 11: Case study 2

Write schema for command print_bad_customers.

Reason: Practically, a comprehension set will be used to answer this problem. However, the comprehension set cannot be translated properly to prolog implementation. Therefore this question is not taken into consideration.

Exercise 12: Case study 2

Ammend video shop state schema so that it can handle multiple copies of videos.

Reason: This question is very general where the type of the variables are not determined. In fact, choosing a suitable type for the system variables is one of the task.

Exercise 13: Case study 2

Ammend the whole system so that it can handle multiple copies of videos.

Reason: This problem is same as above.

Exercise 14: Case study 3

Write a schema for the command Add_waiting_vehicles defined as follows: The command will be given as input to two ferry references. The vehicles waiting for the first ferry will be added to those waiting for the second ferry. An error should be reported if either ferry does not exist, or if the combined set is too big for the new ferry.

Reason: During the preparation for the test data, we found out that the predicate comma is not stable, which we decide to left this question from the assessment.

Exercise 15: Case study 4

Construct a Healthy schema which takes a person's name as input and compares the height with the weight of that person to report either overweight, underweight or OK. You should include error-handling schemas.

Reason: This question cannot be assessed by the system because it is not very clear. The question did not specify the ratio of height and weight in order to produce a

result of overweight, underweight or OK. This will give a problem when we set the test case.

Exercise 16: Case study 4

Amend the Height_and_Weight example to take both height and weight as two integers representing the feet and inches, and stones and pounds, respectively, and to check the validity of the given heights and weights.

Reason: The students were asked to write the type for the system variables, and to write appropriate predicates to fulfill the rest of the problem. This question cannot be assessed by the system because the type of the system variables are not determined.

References

1. D. Rann, J. Turner, and J. Whitworth, *Z: A Beginner's Guide*, Chapman & Hall, 1994.
2. E. Burke and E. Foxley, *Logic and its Applications*, Prentice Hall Europe, 1996.

Appendix E : Score

Q1	h1	h2	h3	system
s1	100	100	100	98
s2	100	100	100	99
s3	100	100	100	98
s4	100	100	100	99
s5	100	100	100	99
s6	100	100	100	98
s7	100	100	100	98
s8	100	100	100	98
s9	100	0	0	30
s10	100	100	100	99

Q2	h1	h2	h3	system
s1	100	100	100	97
s2	100	80	100	97
s3	100	80	100	97
s4	100	80	100	97
s5	65	20	50	2
s6	95	80	100	97
s7	75	60	75	88
s8	100	80	100	97
s9	100	80	100	97
s10	80	40	75	97
s11	100	80	100	97

Q3	h1	h2	h3	system
s1	100	100	100	97
s2	100	100	100	97
s3	90	80	75	98
s4	100	100	100	97
s5	100	100	100	97
s6	100	100	100	97
s7	100	100	100	97
s8	100	100	100	97
s9	100	100	100	97
s10	100	100	100	97
s11	100	100	100	97

Q4	h1	h2	h3	system
s1	80	100	75	86
s2	80	60	55	3
s3	60	60	55	47
s4	80	100	75	86
s5	80	100	75	86
s6	80	100	30	35
s7	60	60	55	47
s8	80	100	75	86
s9	80	60	75	85
s10	50	60	55	2
s11	80	100	75	28

Appendix E : Score

Q5	h1	h2	h3	system
s1	100	100	100	97
s2	100	100	100	97
s3	100	100	100	97
s4	100	100	100	97
s5	100	100	100	97
s6	100	100	100	97
s7	100	100	100	97
s8	100	100	100	97
s9	100	100	100	97
s10	100	100	100	97
s11	100	100	100	97

Q6	h1	h2	h3	system
s1	100	80	79	93
s2	50	40	42	43
s3	50	40	41	17
s4	70	60	42	33
s5	70	60	45	2
s6	70	60	39	44
s7	70	60	47	44
s8	70	60	47	44
s9	50	40	42	2
s10	45	30	42	2

Q7	h1	h2	h3	system
s1	60	60	71	77
s2	80	60	54	73
s3	80	60	50	75
s4	75	40	46	72
s5	75	60	50	72
s6	75	60	43	72
s7	75	60	50	74
s8	45	40	43	73
s9	75	60	43	71
s10	50	40	34	67

Q8	h1	h2	h3	system
s1	100	100	100	97
s2	60	60	64	64
s3	65	60	50	48
s4	70	40	64	64
s5	65	60	50	47
s6	55	80	50	42
s7	20	20	7	25
s8	50	40	50	45
s9	30	20	14	26
s10	50	80	50	81

Appendix E : Score

Q9	h1	h2	h3	system
s1	80	80	67	97
s2	100	100	100	97
s3	100	100	100	97
s4	100	100	100	97
s5	80	80	67	97
s6	100	100	100	97
s7	100	100	100	97
s8	100	100	100	49
s9	20	100	50	77
s10	100	100	100	97
s11	100	100	100	97
s12	100	100	100	97
s13	100	100	100	97

Q10	h1	h2	h3	system
s1	100	100	100	98
s2	55	100	47	0
s3	100	100	100	74
s4	65	100	100	97
s5	100	100	80	75
s6	100	100	100	98
s7	100	100	100	98
s8	75	60	51	0
s9	50	100	100	26
s10	70	100	100	98
s11	100	100	100	98
s12	100	100	100	98
s13	100	100	100	97

Appendix E : Grade

Q1	h1	h2	h3	system
s1	5	5	5	5
s2	5	5	5	5
s3	5	5	5	5
s4	5	5	5	5
s5	5	5	5	5
s6	5	5	5	5
s7	5	5	5	5
s8	5	5	5	5
s9	5	1	1	1
s10	5	5	5	5

Q2	h1	h2	h3	system
s1	5	5	5	5
s2	5	5	5	5
s3	5	5	5	5
s4	5	5	5	5
s5	4	1	3	1
s6	5	5	5	5
s7	5	4	5	5
s8	5	5	5	5
s9	5	5	5	5
s10	5	2	5	5
s11	5	5	5	5

Q3	h1	h2	h3	system
s1	5	5	5	5
s2	5	5	5	5
s3	5	5	5	5
s4	5	5	5	5
s5	5	5	5	5
s6	5	5	5	5
s7	5	5	5	5
s8	5	5	5	5
s9	5	5	5	5
s10	5	5	5	5
s11	5	5	5	5

Q4	h1	h2	h3	system
s1	5	5	5	5
s2	5	4	3	1
s3	4	4	3	2
s4	5	5	5	5
s5	5	5	5	5
s6	5	5	1	1
s7	4	4	3	2
s8	5	5	5	5
s9	5	4	5	5
s10	3	4	3	1
s11	5	5	5	1

Appendix E : Grade

Q5	h1	h2	h3	system
s1	5	5	5	5
s2	5	5	5	5
s3	5	5	5	5
s4	5	5	5	5
s5	5	5	5	5
s6	5	5	5	5
s7	5	5	5	5
s8	5	5	5	5
s9	5	5	5	5
s10	5	5	5	5
s11	5	5	5	5

Q6	h1	h2	h3	system
s1	5	5	5	5
s2	3	2	2	2
s3	3	2	2	1
s4	5	4	2	1
s5	5	4	2	1
s6	5	4	1	2
s7	5	4	2	2
s8	5	4	2	2
s9	3	2	2	1
s10	2	1	2	1

Q7	h1	h2	h3	system
s1	4	4	5	5
s2	5	4	3	5
s3	5	4	3	5
s4	5	2	2	5
s5	5	4	3	5
s6	5	4	2	5
s7	5	4	3	5
s8	2	2	2	5
s9	5	4	2	5
s10	3	2	1	4

Q8	h1	h2	h3	system
s1	5	5	5	5
s2	4	4	4	4
s3	4	4	3	2
s4	5	2	4	4
s5	4	4	3	2
s6	3	5	3	2
s7	1	1	1	1
s8	3	2	3	2
s9	1	1	1	1
s10	3	5	3	5

Appendix E : Grade

Q9	h1	h2	h3	system
s1	5	5	4	5
s2	5	5	5	5
s3	5	5	5	5
s4	5	5	5	5
s5	5	5	4	5
s6	5	5	5	5
s7	5	5	5	5
s8	5	5	5	2
s9	1	5	3	5
s10	5	5	5	5
s11	5	5	5	5
s12	5	5	5	5
s13	5	5	5	5

Q10	h1	h2	h3	system
s1	5	5	5	5
s2	3	5	2	1
s3	5	5	5	5
s4	4	5	5	5
s5	5	5	5	5
s6	5	5	5	5
s7	5	5	5	5
s8	5	4	3	1
s9	3	5	5	1
s10	5	5	5	5
s11	5	5	5	5
s12	5	5	5	5
s13	5	5	5	5

Appendix E : Grouped Frequency

Marker : h1

Question	70-100	60-69	50-59	40-49	0-39
1	10	0	0	0	0
2	10	1	0	0	0
3	11	0	0	0	0
4	8	2	1	0	0
5	11	0	0	0	0
6	6	0	3	1	0
7	7	1	1	1	0
8	2	3	3	0	2
9	12	0	0	0	1
10	10	1	2	0	0

Marker : h2

Question	70-100	60-69	50-59	40-49	0-39
1	9	0	0	0	1
2	8	1	0	1	1
3	11	0	0	0	0
4	6	5	0	0	0
5	11	0	0	0	0
6	1	5	0	3	1
7	0	7	0	3	0
8	3	3	0	2	2
9	13	0	0	0	0
10	12	1	0	0	0

Marker : h3

Question	70-100	60-69	50-59	40-49	0-39
1	9	0	0	0	1
2	10	0	1	0	0
3	11	0	0	0	0
4	6	0	4	0	1
5	11	0	0	0	0
6	1	0	0	8	1
7	1	0	4	4	1
8	1	2	5	0	2
9	10	2	1	0	0
10	11	0	1	1	0

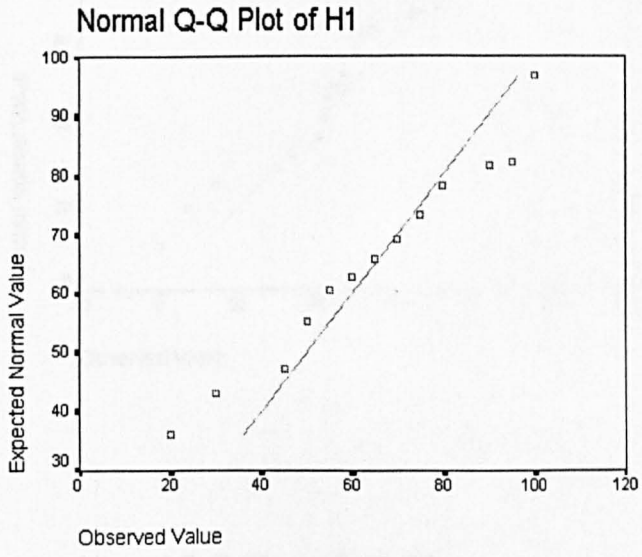
Marker : system

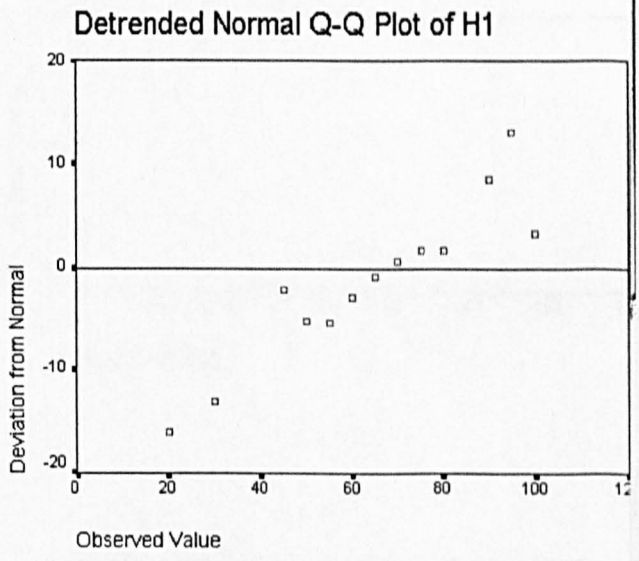
Question	70-100	60-69	50-59	40-49	0-39
1	9	0	0	0	1
2	10	0	0	0	1
3	11	0	0	0	0
4	5	0	0	2	4
5	11	0	0	0	0
6	1	0	0	4	5
7	9	1	0	0	0
8	2	2	0	4	2
9	12	0	0	1	0
10	10	0	0	0	1
Total	80	3	0	11	14

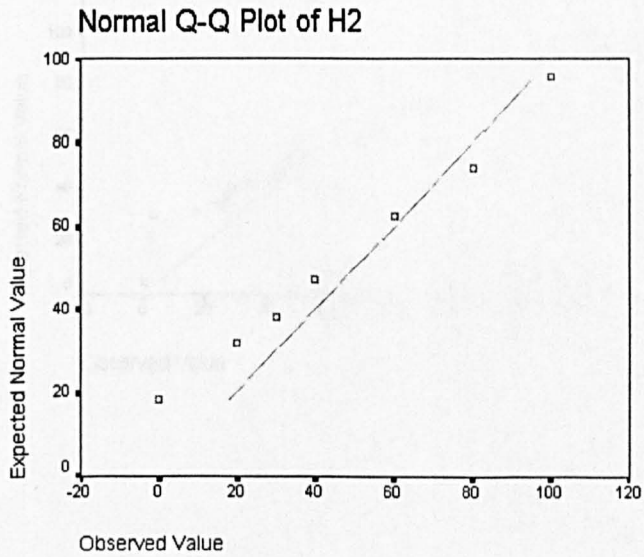
TEXT CUT OFF IN THE ORIGINAL

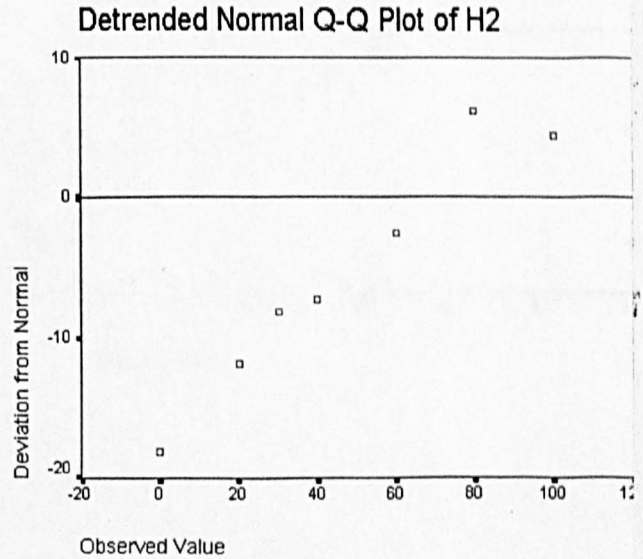
Appendix F
Normal Q-Q Plot of Data

Data type : Score

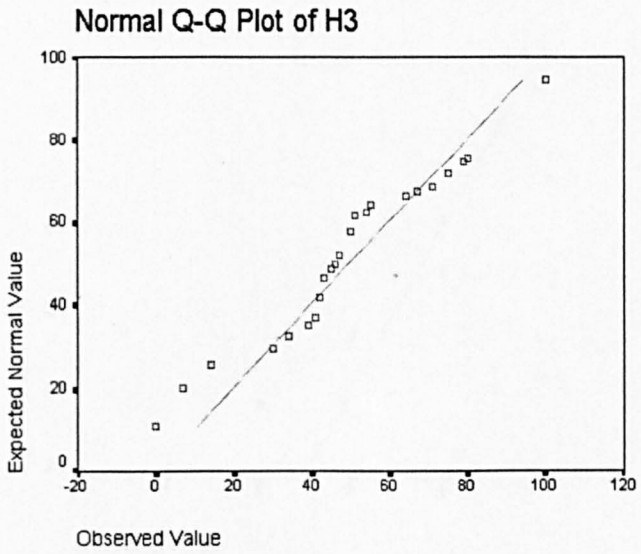


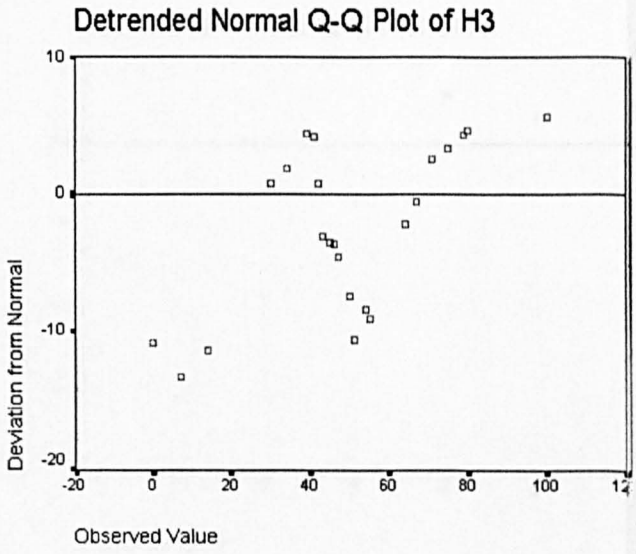


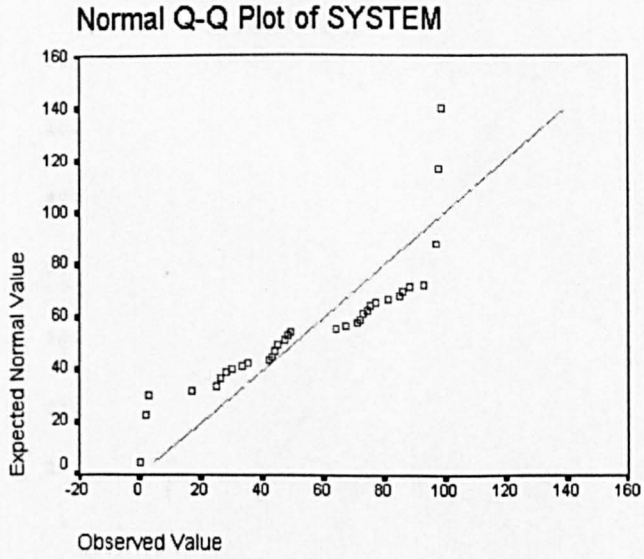


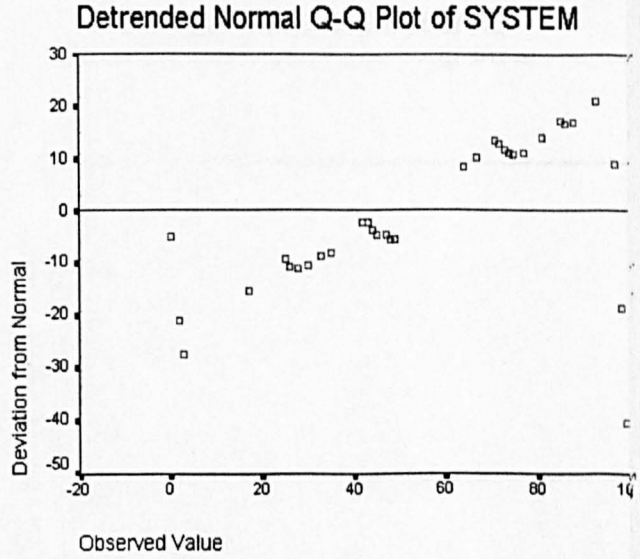


Data type : Score

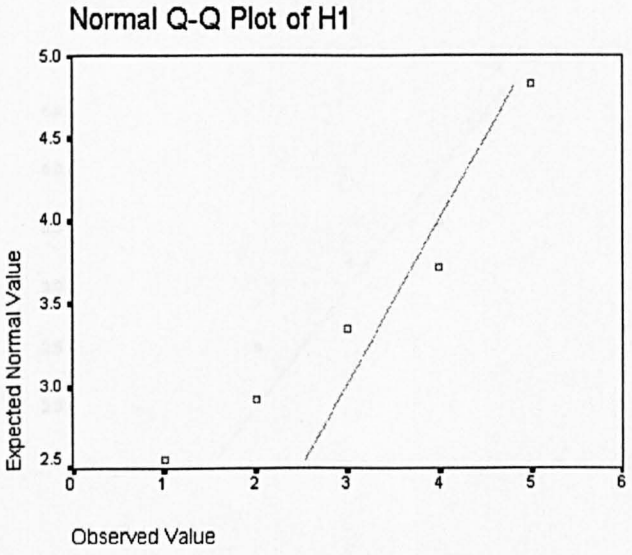


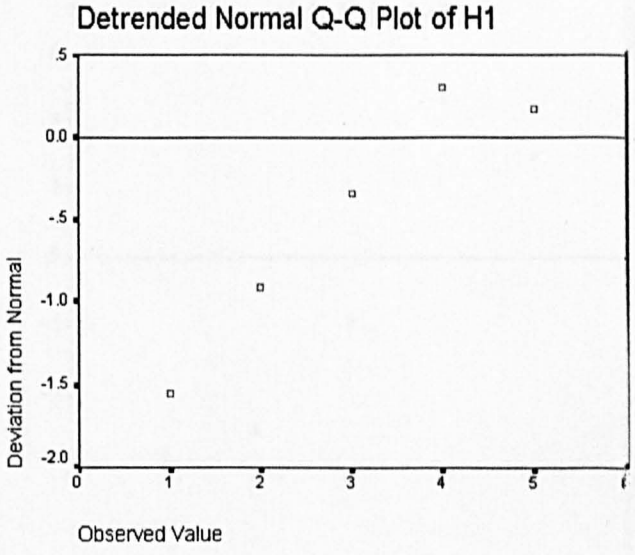


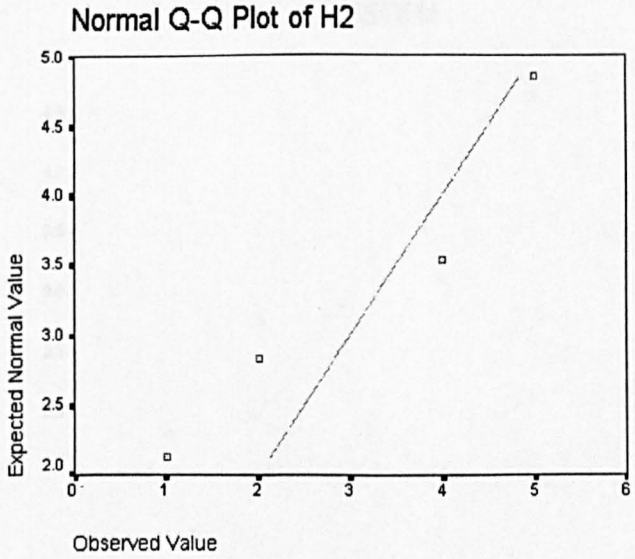


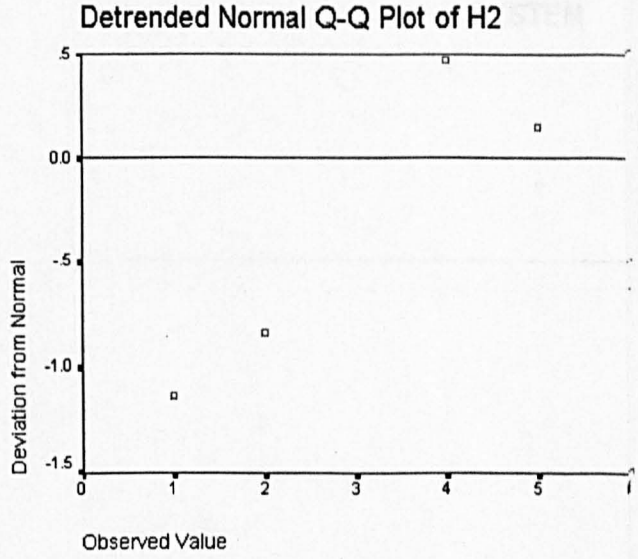


Data type : Grade

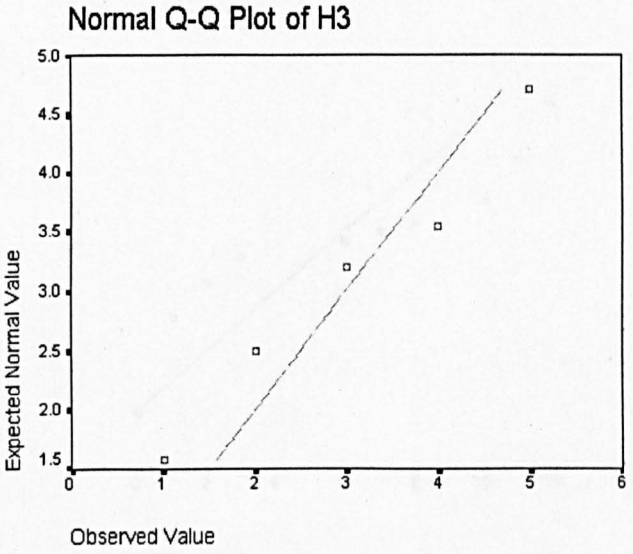


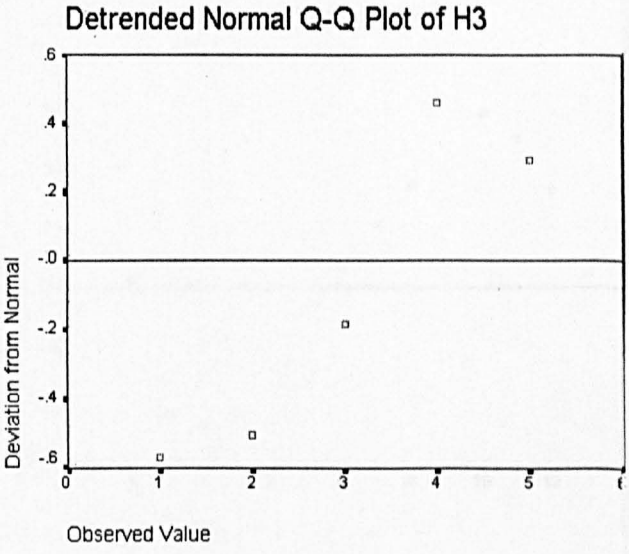


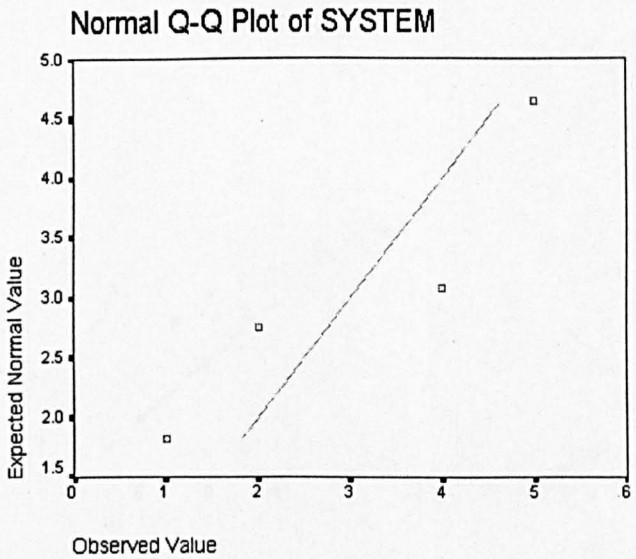


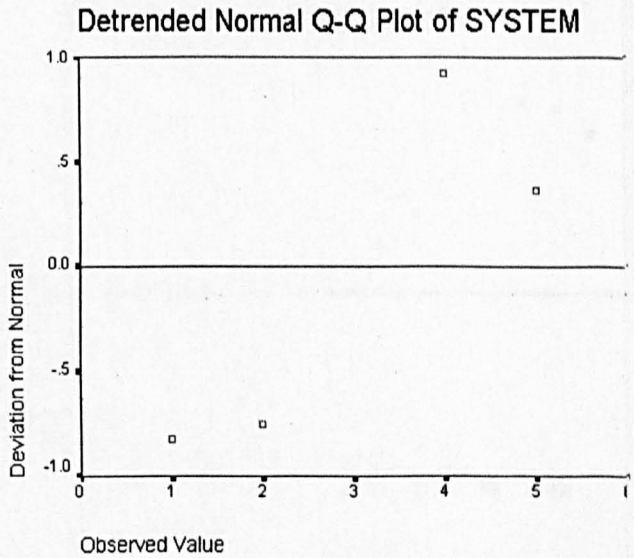


Data type : Grade

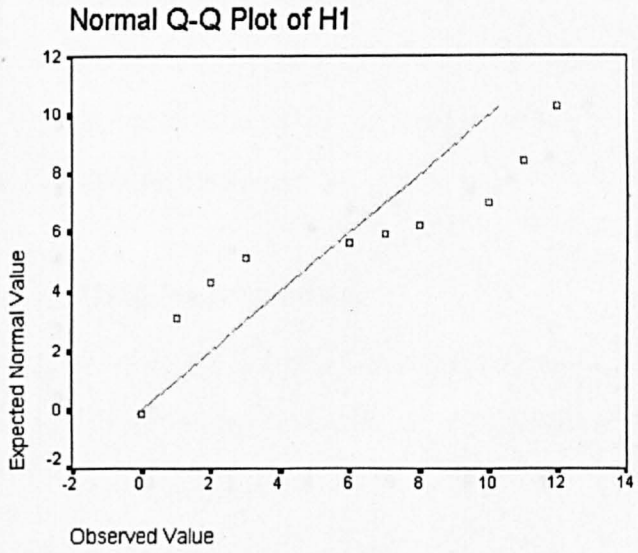


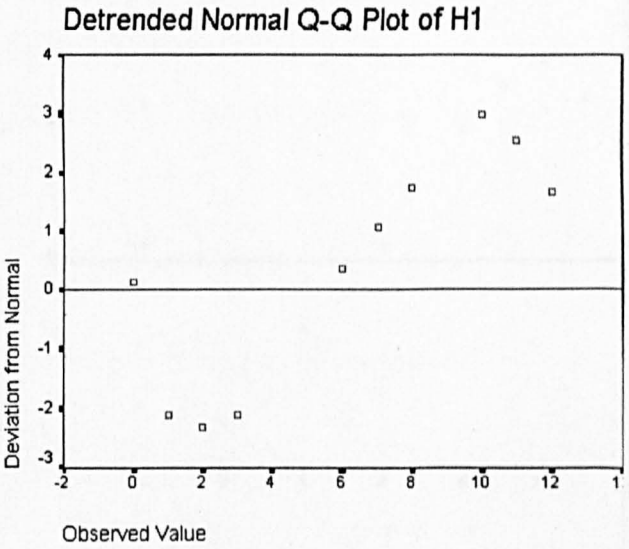


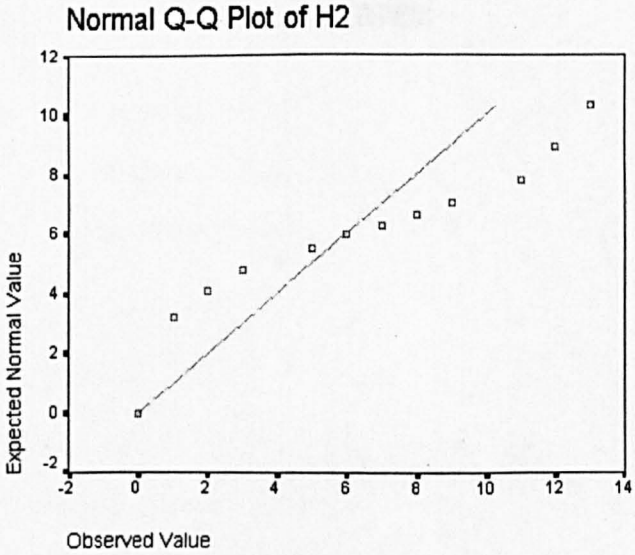


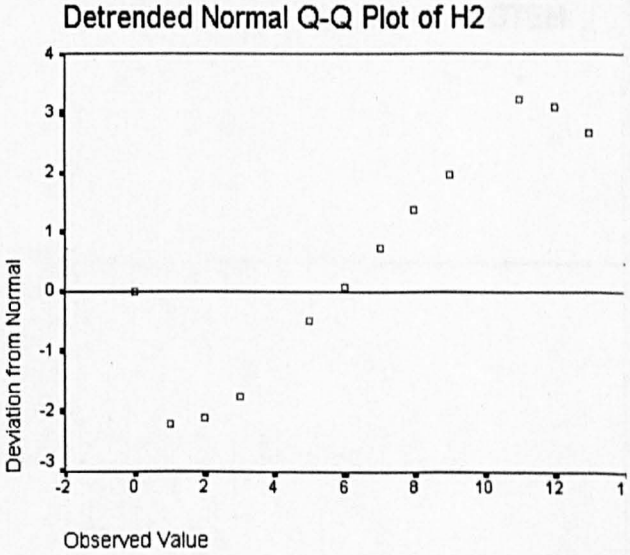


Data type : Grouped frequency

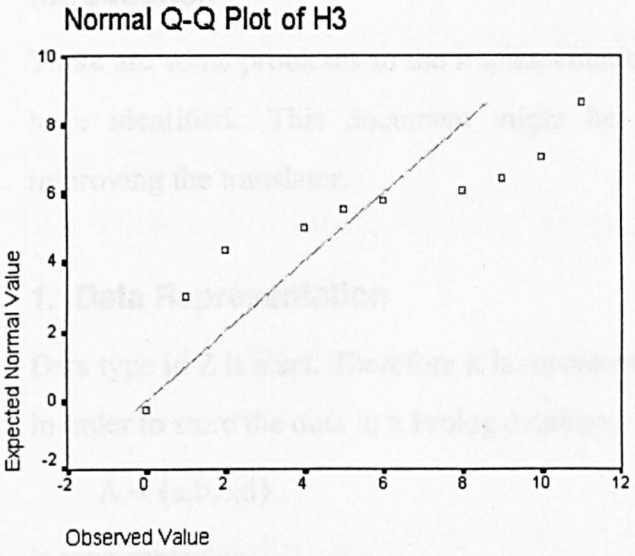


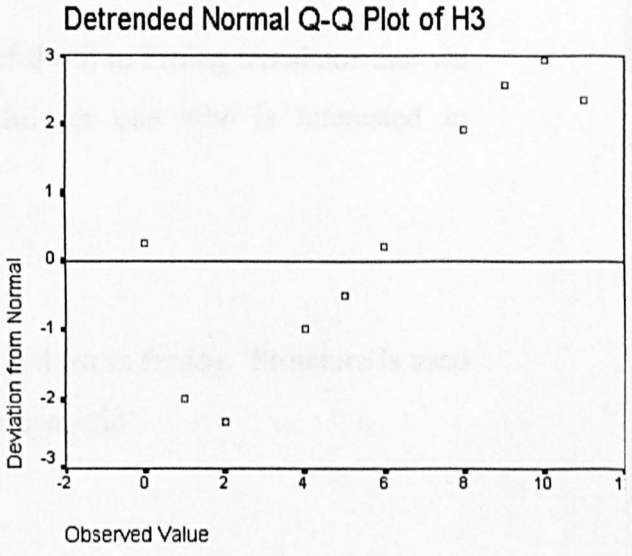


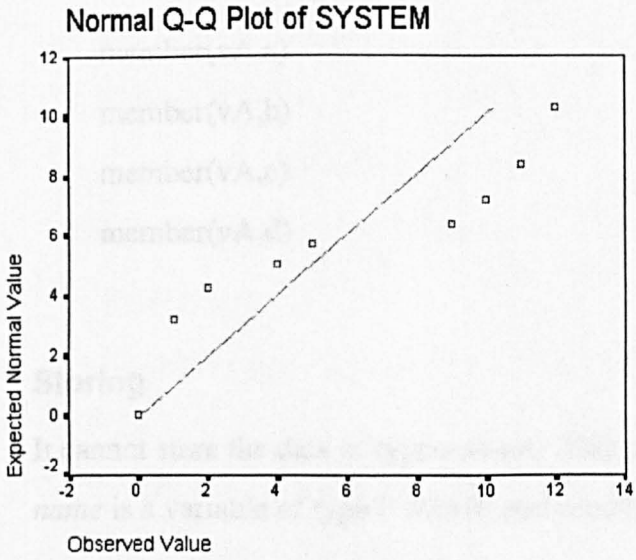


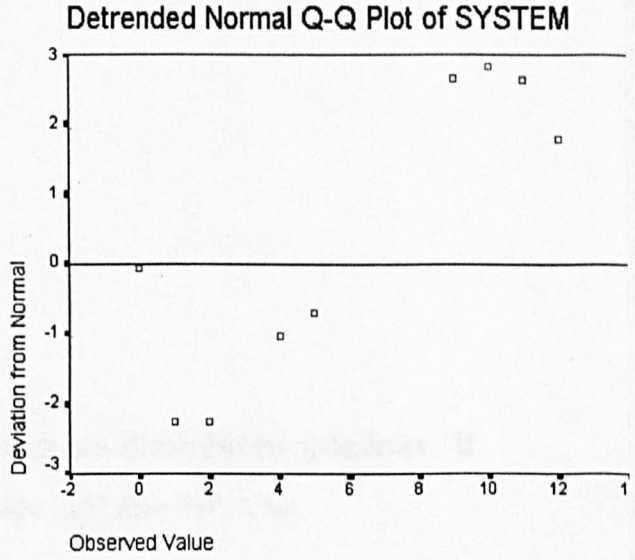


Data type : Grouped frequency









Appendix G
Comments on Prolog implementation by *zp*

Introduction

There are some problems in the implementation of the Z to Prolog translator that we have identified. This document might be useful for one who is interested in improving the translator.

1. Data Representation

Data type in Z is a set. Therefore it is represented as a list in Prolog. Structure is used in order to store the data in a Prolog database. For example

A = {a,b,c,d}

is represented as

A = [a,b,c,d]

and is stored (during runtime) in Prolog database as

member(vA,a)

member(vA,b)

member(vA,c)

member(vA,d)

Storing

It cannot store the data of type non-set. This problem can be explained as follows. If *name* is a variable of type **P** *NAME* and contains data {*ali*,*abu*,*eric* }, i.e

name={*ali*,*abu*,*eric* },

the information can be stored as a structure of

member(vname,ali)

member(vname,abu)

member(vname,eric)

in prolog. However, if *name* is a variable of type *NAME* and associated with *ali*, i.e

name=ali,

it does not provide the structure to store the information *ali*.

Tuple and Set

It is unable to differentiate between tuple and set when using list. A set of sets, such as

A = { {a}, {b}, {c,d} }

is represented as a list of

[[a],[b],[c,d]]

A set of tuples for example

B = { (a,b),(c,d),(e,f)}

is represented as a list in Prolog by

[[a,b],[c,d],[e,f]]

The problem is that if we have a set

C = { {a,b}, {c,d}, {e,f} }

it will also be presented in Prolog as

[[a,b],[c,d],[e,f]]

As a result we cannot differentiate both of them.

2. Operation Representation

Comprehensive set

The comprehensive set is translated into Prolog by using the built-in predicate `setof`. The problem is that predicate `setof` is unable to handle backtracking. For example,

$$\{a \mid a \in A; a > 0\}$$

is translated into

```
example1(L) :-
     setof(Va, expl(Va), L).
expl(Va) :-
     member(setA,Va),
     greater(Va,0).
```

Normally if one of the goals under `expl` failed, there is a possibility of backtracking execution. If this happen, predicate `setof` will fail, and consequently predicate `example1` will also fail.

Logical operation : IMPLIES

The design of a library for function IMPLIES is not proper. Basically, predicate $A \Rightarrow B$ will be translated into Prolog as

```
exp(A,RA),
exp(B,RB),
imp(RA,RB,R).
```

RA and **RB** are the results that produce either true or false when executing expression **A** and **B** respectively. **R** is a result of either true or false depending on value **RA** and **RB.** The logical aspect of the predicate is correct. However, the executional aspect of the predicate is not proper. Expression **B** will be carried out regardless whether expression **A** is true or not. For example,

$$A > 7 \Rightarrow B = 4$$

$$A \leq 7 \Rightarrow B=6$$

will be translated into Prolog as

```
1:  greater(A,7,RA1),
2:  equal(B,4,RB1),
3:  imp(RA1,RB1,R1),
4:  lequal(A,7,RA2),
5:  equal(B,6,RB2),
6:  imp(RA2,RB2,R2).
```

Prolog will execute all the lines sequentially. At the line 1, if A > 7 then **RA1** is equal to *true*. The result is *false* otherwise. Regarding **B,** it will always be assigned with 4 because the line 2 will always be executed regardless whether predicate A > 7 true or not. Therefore at the line 5, **RB2** will always false because **B** is not equal to 6. **B** is equal to 4.

Quantifiers

Some quantified expression is not translated properly. The error normally found is:

forall(V999,exp1(V329),)

when it is supposed to be:

forall(V999,exp1(V329),dot1(V329))

A program has been made to correct this particular error.

Another executional problem is when a schema, which consists of quantified expression, is included in another schema. Before any schema is translated, it will first be expanded to remove all the included schemas. This expanded schema is then translated into Prolog. The algorithm involves is

```
for all included schema S
    getcode
    addcodebefore
```

```
for all included schema S
     getcode
     if sign (S) is DELTA
          addcodeafter
     else if sign (S) is INVARIANT
          addcodeafter
          addinvariant
```

During the expansion, the quantified expression has no code for after operation ('primed' variable). Instead it will be written as a code before operation in a location where it should be a code after operation. It seems that the algorithm executed for quantified expression is as below:

```
for all included schema S
     getcode
     addcodebefore
for all included schema S
     getcode
     if sign (S) is DELTA
          addcodebefore
     else if sign (S) is INVARIANT
          addcodebefore
          addinvariant
```

This problem can be illustrated as the following.

Schema 1

$$A : \mathbb{Z}$$
$$B : \mathbb{Z}$$

$$A = B$$
$$\forall a : A; b : B \bullet a > 0 \land b > 0$$

Schema 2

Δ*Schema* 1
$C : \mathbb{Z}$

$C=A$

Schema1 is included in schema2. Expanded schema for Schema2 is

Schema 2

$A : \mathbb{Z}$
$B : \mathbb{Z}$
$C : \mathbb{Z}$
$A' : \mathbb{Z}$
$B' : \mathbb{Z}$

$A=B$
$\forall a : A ; b : B \bullet a>0 \land b>0$
$C=A$
$A'=B'$
$\forall a : A ; b : B \bullet a>0 \land b>0$

Note the second quantified expression. It should be

$$\forall a:A';b:B' \bullet a>0 \land b>0$$

To correct this, we should focus on the type and syntax checker `zc`, as well as the Z expander, `ze`. Expression quantifiers such as Σ Ω μ and λ also face the same problem.

3. The Z prolog library

Small corrections have been made to the set relations, the logical expressions and the list operations in the Z definition library.

Besides, we also detected the existence of problems in predicate `comma` and predicate `docard`.