

Sheffield Hallam University

An editor and transformation system for a Z animation case tool.

BUCKBERRY, Graham R.

Available from the Sheffield Hallam University Research Archive (SHURA) at:

<http://shura.shu.ac.uk/19404/>

A Sheffield Hallam University thesis

This thesis is protected by copyright which belongs to the author.

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Please visit <http://shura.shu.ac.uk/19404/> and <http://shura.shu.ac.uk/information.html> for further details about copyright and re-use permissions.

LEARNING CENTRE
CITY CAMPUS, POND STREET,
SHEFFIELD, S1 1WB.

101 568 276 6



19 DEC 2005

5pm

15 MAR 2006

4:12pm

21 APR 2006

4.09pm

ProQuest Number: 10694285

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10694285

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

An Editor and Transformation System for a
Z Animation CASE Tool

Graham Robert Buckberry

A thesis submitted in partial fulfilment of the requirements of
Sheffield Hallam University
For the degree of Doctor of Philosophy

August 1999



LEVEL 1

Acknowledgements

I would like to express my thanks to Mr Ian Morrey and Professor Jawed Siddiqi of the department of Computing and Management Sciences, Sheffield Hallam University, for their advice, guidance and enthusiasm throughout the course of this project.

I would also like to acknowledge the work of Richard Hibberd in the development of the ZAL animation environment, and Paul Parry for his work in the development of the ViZ visualisation system, both of which are referenced in this thesis.

In addition, I would like to express my gratitude to Siemens Communications Limited, and in particular Mr Roger Andrews, Engineering Manager, for the support the company has given me in the completion of this work.

No one can aspire to complete a work of this magnitude without the support of his or her family. I have indeed been fortunate to enjoy the love and encouragement of my wife Lorraine throughout this project, without whom I am sure I would not have completed this work. Thanks is too small a word.

Graham R. Buckberry. B.Eng (Hons), M.Sc, C.Eng, MIEE. August 1999.

Microsoft, MS and MS-DOS are registered trademarks and Windows is a trademark of the Microsoft Corporation. IBM and PC/AT are registered trademarks and PC/XT is a trademark of International Business Machines Corporation.

Table of Contents

ACKNOWLEDGEMENTS	2
TABLE OF CONTENTS	4
TABLE OF FIGURES	9
LIST OF TABLES	11
ABSTRACT	12
1. INTRODUCTION AND PROBLEM DEFINITION	13
1.1 The Risks and Costs of Developing Software	14
1.1.1 Developers Under Pressure	16
1.1.2 The Need to Capture What the User Requires	17
1.2 Software Engineering Processes and Practice	18
1.2.1 Software Engineering Practice	20
1.2.2 The Software Lifecycle	20
1.2.3 Identifying Costs in the Waterfall Model	25
1.2.4 The Implications of Software Evolution for Requirements Engineering	26
1.2.5 Improving the Lifecycle Model	28
1.2.6 The Case for Prototyping	29
1.2.7 Rapid Prototyping, Animation and Executable Specifications	31
1.2.8 The Application of CASE Tools	32
1.3 The Objectives of the Research Programme	33
1.3.1 The Project Plan	36
1.4 Thesis Structure	37
2. REQUIREMENTS ENGINEERING AND FORMAL SPECIFICATION	39
2.1 What is a Requirement?	39
2.1.1 Requirements Engineering Methods	41
2.1.2 The Activities within Requirement Engineering	43
2.1.3 Factors Affecting the Requirement Engineering Task	44
2.2 A General Approach to Requirements Engineering	45
2.2.1 Roles and Communication in Requirements Engineering	48
2.2.2 Techniques Promoting User Consultation	48
2.2.3 Techniques Promoting User Participation	50
2.2.4 Techniques Promoting Stakeholder Participation	52
2.2.5 Techniques Promoting Stakeholder Co-operation	53
2.2.6 The Specification of Requirements	56
2.3 Approaches Based on Formal Methods	57

2.3.1	Formal Specification Techniques	60
2.3.2	The Z Notation	62
2.3.3	Formal Specification and Bridging the Communication Gap	65
2.3.4	Related Research into Executable Specifications	66
2.4	Summary	69
3.	REQUIREMENTS ENGINEERING TOOLS AND PROCESSES	71
3.1	Developing a Requirements Engineering Tools Hierarchy	72
3.1.1	Identifying the Users of Requirements Engineering Tools	75
3.1.2	Identifying Requirements Engineering Problems from the Primary Viewpoints	78
3.1.3	Tools to Control the Relationship Between Requirements and System Costs	79
3.1.4	Tools Which Capture Requirements for System Reliability and Maintainability	80
3.1.5	Tools Which Capture Requirements Associated with System Functionality	81
3.1.6	Tools for Specification Validation	82
3.1.7	A Requirements Engineering Tools Hierarchy	84
3.1.8	Requirements Engineering Tools Limitations	86
3.2	Developing Requirements for a Requirement Engineering Tool	86
3.2.1	Developing Requirements for a Tool to Support the Representation Process	88
3.2.2	Introducing Formalism Within the Representation Process	90
3.2.3	Justification for the Use of the Z notation	90
3.2.4	Product Requirements for a Tool to Support the Representation Process	92
3.2.5	Developing Requirements for a Tool to Support the Specification Process	92
3.2.6	Understanding How Specifications are Developed	95
3.2.7	The Usability Phase	98
3.2.8	The Functional Phase	99
3.2.9	The Resolution Phase	99
3.2.10	Conclusions from the Study of Specification Development	101
3.2.11	Managing the Interfaces Between Specification Components	102
3.2.12	Assessing the Impact of Formal Methods on the Specification Process	103
3.2.13	Product Requirements for a Tool to Support the Specification Process	106
3.2.14	Developing Requirements for a Tool to Support the Agreement Process.	107
3.2.15	The Semantic and Implementation Gaps	108
3.2.16	Product Requirements for a Tool to Support the Agreement Process	110
3.3	A Process for Requirements Engineering Using TranZit, ZAL and ViZ	110
3.3.1	The REALiZE Process	113
3.4	Summary	115
4.	REALISATION OF THE TRANZIT EDITOR AND ANALYSER SUBSYSTEM	117
4.1	Research and Development of the TranZit User Interface	118
4.1.1	The TranZit User Interface Design	120
4.1.2	The TranZit Main Editor Window	122
4.1.3	The Use of Object-Orientation to Support the Capture of Z specifications	124
4.1.4	Accommodating the Learning Potential of the User	125
4.1.5	Considerations in Designing the GUI for the TranZit Analyser Subsystem	128
4.1.6	The User Interface to the TranZit Transformation Engine (TTE)	131
4.1.7	Evaluating the TranZit User Interface	133
4.2	Research and Development of the TranZit Syntax Analyser	133
4.2.1	The Parsing Problem	134

4.2.2	Definitions for Languages and Grammars	137
4.2.3	Language Classes	139
4.2.4	Grammars and Automata	140
4.2.5	Grammars and Ambiguity	143
4.2.6	Developing a Parser for the Z Notation Syntax	145
4.2.7	Applying Iteration and Factoring Transformations to the Z Notation Grammar	147
4.2.8	Recursive Descent Techniques	152
4.2.9	Recovery From Errors	155
4.3	Research and Development of the TranZit Type Checker	156
4.3.1	Rationale and Design Criteria for the TranZit Schema ObjectBase	156
4.3.2	Implementing the TranZit Schema ObjectBase	159
4.3.3	Realising a Type Checker for the Z notation	164
4.3.4	Representing Types in the Z notation	165
4.3.5	Performing the Type Checking function	167
4.3.6	Other Semantic Actions	171
4.4	Summary	173
5.	RESEARCH AND DEVELOPMENT OF THE TRANZIT TRANSFORMATION ENGINE	175
5.1	The Rationale for Transformation	175
5.1.1	Constructing an Executable Representation of a Specification	177
5.1.2	Evolution of the Z Animation Language (ZAL)	179
5.1.3	Development of the ZAL Grammar	184
5.1.4	Evolution of the TranZit Transformation Engine (TTE)	186
5.1.5	Requirements for the TranZit Transformation Engine	187
5.1.6	Intermediate Languages	188
5.1.7	Accessibility of the Executable Representation	189
5.1.8	Ensuring Transformation Correctness	191
5.2	Realisation of the TranZit Transformation Engine	192
5.2.1	TranZit Transformation System Architecture	194
5.2.2	The Production System	195
5.2.3	The Polish Conversion Engine	195
5.2.4	The Computability Analyser	196
5.2.5	Animation Support	197
5.2.6	Support for Shown Variables	197
5.2.7	Resolution of Implicit Schemas	198
5.2.8	The Format Engine	199
5.3	Non-Computable Aspects of Specification Languages	201
5.3.1	Computability and the Z Notation	203
5.3.2	A Strategy for Dealing With Non-Computable Clauses in Z	204
5.3.3	Example: IsAPerfectSquare	205
5.3.4	Adding Constraints to Non-Computable Clauses	206
5.3.5	The Computability Analyser: Identifying Enumeration Functions	210
5.3.6	An Eclectic Strategy: The TranZit Transformation Assistant	212
5.4	Summary	215
6.	TRANZIT SYSTEM TESTING, EVALUATION AND CASE STUDIES	217
6.1	The Testing of TranZit	217
6.1.1	Unit Testing	219

6.1.2	Integration Testing	220
6.1.3	Acceptance Testing	222
6.2	Assessment of the Usability of TranZit	225
6.2.1	Analysis of Feedback from User Questionnaires	225
6.3	Comparison of TranZit and other Requirements Engineering Tools	227
6.3.1	The Use of Formalism	227
6.3.2	Tools Platforms	228
6.3.3	Comparison of the TranZit and Formaliser Tools	229
6.3.4	Comparison of the TranZit and ZFDSS Tools	229
6.4	Case Study I: A Library System	233
6.4.1	Z Specification Development	233
6.4.2	Capturing in TranZit and Transformation to ZAL	235
6.4.3	Animation in the ZAL Environment	240
6.4.4	Creating Candidate Data	241
6.4.5	Animating the Library Specification	242
6.4.6	Discussion	247
6.5	Case Study II: The Telephone Network	248
6.5.1	Z Specification Development	248
6.5.2	Capturing in TranZit and Transformation into ZAL	251
6.5.3	Animation in the ZAL Environment	255
6.5.4	Creating Candidate Data	256
6.5.5	Executing the Telephone Network Animation	257
6.5.6	Discussion	263
6.6	Summary	264
7.	RESULTS AND CONCLUSIONS	266
7.1	Review of Achievement Against Research Programme Objectives	266
7.2	Opportunities for Further Research Work	269
7.3	General Conclusions	272
	BIBLIOGRAPHY	274
	APPENDIX I: LL(K) GRAMMAR FOR THE Z NOTATION	291
	APPENDIX II: CONTEXT-FREE GRAMMAR OF THE ZAL LANGUAGE	298
	APPENDIX III : A REVIEW OF CURRENT REQUIREMENTS ENGINEERING TOOLSETS	303
	APPENDIX IV: TRANZIT USER QUESTIONNAIRE RESULTS	310
	Appendix IV-1: Population Statistics	310
	Appendix IV-II: About Requirements Engineering	311

Appendix IV-III: About the Z Notation	313
Appendix IV-IV: About TranZit	315
APPENDIX V: PUBLICATION HISTORY	317
APPENDIX VI: GLOSSARY OF ABBREVIATIONS	318

Table of Figures

<i>Figure 1-1: Developers under Pressure</i>	16
<i>Figure 1-2: Study Phase Costs as a Percentage of Development Costs</i>	17
<i>Figure 1-3: Waterfall Model of Software Systems Development</i>	24
<i>Figure 1-4: Prototyping Lifecycle Model</i>	29
<i>Figure 2-1: Categories of Requirements Engineering Methods</i>	41
<i>Figure 2-2: Requirements Engineering Process Framework</i>	46
<i>Figure 2-3: Kano's Model of Customer Requirements</i>	55
<i>Figure 2-4: Example Z Specification</i>	64
<i>Figure 3-1: Requirements Engineering Tools Hierarchy</i>	85
<i>Figure 3-2: Pohl's Three Dimensions of Requirements Engineering</i>	87
<i>Figure 3-3: Considerations in the Specification Process</i>	93
<i>Figure 3-4: Industrial Specification Process</i>	97
<i>Figure 3-5: The three phases of Specification Production</i>	101
<i>Figure 3-6: The REALiZE Process</i>	113
<i>Figure 3-7: The Logical Interfaces between TranZit, ZAL and ViZ</i>	115
<i>Figure 4-1: Hierarchy of TranZit Features</i>	122
<i>Figure 4-2: TranZit Main Editor Window</i>	123
<i>Figure 4-3: TranZit Open Schema Dialog</i>	125
<i>Figure 4-4: TranZit Symbol Selection Dialog</i>	127
<i>Figure 4-5: TAS Control Dialog</i>	129
<i>Figure 4-6: TAS Results Window</i>	130
<i>Figure 4-7: TAS Type Mismatch Summary Example</i>	130
<i>Figure 4-8: Example Transformation Output Window</i>	132
<i>Figure 4-9: Example Syntax Tree</i>	134
<i>Figure 4-10: Example BNF Notation</i>	136
<i>Figure 4-11: Expansion by the Production System</i>	136
<i>Figure 4-12: TranZit Schema Objectbase Concept</i>	157
<i>Figure 4-13: TranZit Schema Objectbase Internal Structure</i>	160
<i>Figure 4-14: Local Declarations ObjectArray Structure</i>	164
<i>Figure 4-15: Representing Types</i>	166
<i>Figure 4-16: Example Interactions within the Type Checker</i>	170
<i>Figure 5-1 : Comparing Executability and Expressibility</i>	178
<i>Figure 5-2: Example Specification with Corresponding ZAL Representation</i>	184
<i>Figure 5-3: The Interface between TranZit and the ZAL Animation Environment</i>	186
<i>Figure 5-4: Mapping Between Semantic Universes</i>	192
<i>Figure 5-5: TranZit Transformation System Architecture</i>	194
<i>Figure 5-6: Set Show Variable Dialog</i>	198
<i>Figure 5-7: The Transformation System Dialog</i>	200
<i>Figure 5-8: Typical TranZit Window Arrangement for Transformation Auditing</i>	201
<i>Figure 5-9: TranZit Transformation Assistant Dialog</i>	213
<i>Figure 6-1: Integration Test Strategies</i>	220
<i>Figure 6-2: Capturing the Library Specification in TranZit</i>	236
<i>Figure 6-3: Screen Dump from TAS for Library Specification</i>	236
<i>Figure 6-4: Screen Dump Showing TTE Output for Library Specification</i>	237
<i>Figure 6-5: Using TranZit and ZAL to Animate the Library Specification</i>	240
<i>Figure 6-6: Creating Candidate Data for the Library Animation</i>	242
<i>Figure 6-7: Executing the Library Animation</i>	243
<i>Figure 6-8: ZAL Execution Feedback Window on Executing Schema Borrow</i>	244
<i>Figure 6-9: Provocative Testing of the Library Animation</i>	245
<i>Figure 6-10: Executing the Revised Library Animation</i>	246
<i>Figure 6-11: Output from ZAL on Executing the Revised Library Animation</i>	246
<i>Figure 6-12: TranZit Transformation Assistant Request Dialog for Conns0</i>	252
<i>Figure 6-13: TranZit Transformation Assistant Request Dialog for Conns1</i>	253
<i>Figure 6-14: TranZit Transformation Assistant Request Dialog for Variable C</i>	253

<i>Figure 6-15: Executing the Telephone Network Animation</i>	256
<i>Figure 6-16: Creating Candidate Data for the Telephone Network Animation</i>	257
<i>Figure 6-17: Result of Executing Schema TN</i>	257
<i>Figure 6-18: Result of Executing Schema TN with Revised Candidate Data</i>	258
<i>Figure 6-19: Result of Executing Schema TN with further Candidate Data Changes</i>	259
<i>Figure 6-20: Result of Executing Schema TN as part of evaluating EfficientTN</i>	259
<i>Figure 6-21: Result of Executing Schema EfficientTN</i>	260
<i>Figure 6-22: ZAL Execution Feedback Window on Executing Schema Call</i>	261
<i>Figure 6-23: Creating Candidate Data for the Hangup Operation</i>	262
<i>Figure 6-24: Result of Executing the Hangup Schema</i>	262

List of Tables

<i>Table 1: Stakeholder Concerns</i>	77
<i>Table 2: Product Requirements for a Tool to support the Representation Process</i>	92
<i>Table 3: Product Requirements for a Tool to support the Specification Process</i>	106
<i>Table 4: Product Requirements for a Tool to Support the Agreement Process</i>	110
<i>Table 5: Table of ZAL Operators</i>	182
<i>Table 6: Product Requirements Checklist</i>	224
<i>Table 7: Table of Meta-Symbols for the LL(k) Grammar</i>	291
<i>Table 8: Symbols Which Can Be Surrounded by NL Characters</i>	291
<i>Table 9: Table of Terminal Symbols for LL(k) Grammar</i>	297
<i>Table 10: Table of Meta-Symbols for the ZAL Grammar</i>	298
<i>Table 11: Table of Terminal Symbols for the ZAL Grammar</i>	302
<i>Table 12: Review of Requirements Engineering Toolsets</i>	308
<i>Table 13: REALiZE Toolset Capabilities</i>	309
<i>Table 14: Questionnaire Results: User Experience I</i>	310
<i>Table 15: Questionnaire Results: User Experience II</i>	311
<i>Table 16: Questionnaire Results: Opinions on Requirements Engineering I</i>	312
<i>Table 17: Questionnaire Results: Opinions on Requirements Engineering II</i>	312
<i>Table 18: Questionnaire Results: Opinions on Requirements Engineering III</i>	313
<i>Table 19: Questionnaire Results: Opinions on the Z Notation I</i>	313
<i>Table 20: Questionnaire Results: Opinions on the Z Notation II</i>	314
<i>Table 21: Questionnaire Results: Opinions on the Z Notation III</i>	314
<i>Table 22: Questionnaire Results: Opinions on the TranZit Tool I</i>	315
<i>Table 23: Questionnaire Results: Opinions on the TranZit Tool II</i>	315
<i>Table 24: Questionnaire Results: Opinions on the TranZit Tool III</i>	316
<i>Table 25: Questionnaire Results: Opinions on the TranZit Tool IV</i>	316
<i>Table 26: Table of Publications Associated with this Project</i>	317

Abstract

In order to remain competitive, modern systems developers are increasingly under pressure to produce software solutions to complex problems faster and cheaper, whilst at the same time maintaining a high level of quality in the delivered product. One of the key quality measures is the delivery of a system that meets the customer's *requirements*. Failure to meet the customer's requirements may engender significant re-design, which in turn will cost money, delay product introduction and may seriously damage the developer's credibility. For these reasons, the problem of developing a precise and unambiguous statement of requirements for a proposed system is perhaps one of the most challenging problems within software engineering today.

Formal, model-based *specification languages* such as the Z notation have been widely adopted within the context of *requirements engineering*, to provide a vehicle for the development of precise and unambiguous specifications. However, the mathematical foundation upon which these notations are based often makes them unapproachable and difficult to assimilate by a non-specialist reader. The problem then faced is that if the customer cannot understand the semantics of the specification, how can the customer agree that the specification is indeed a true reflection of the requirements for the desired system?

Several researchers have proposed that *rapid prototyping* and *animation* of specifications can be used to increase the customer's understanding of the formal specification. This is achieved by *executing* specification components on candidate data and observing that the behaviour is as expected. However this requires that the original formal specification be reliably *transformed* into a representation capable of being executed within a computer system. To achieve this aim requires the support of *computer-based tools* able to assist the requirements engineer in capturing, manipulating and transforming the formal specification in an efficient and consistent manner.

This thesis describes the research and development of the TranZit tool, which is a Z notation editor, checker and transformation system. TranZit supports the efficient capture and maintenance of Z notation specifications using the Windows™ Graphical User Interface, supported by a suite of powerful language-driven features. In addition TranZit contains a highly integrated and optimised syntax and type checker, combining traditional compiler design techniques with innovative use of object-oriented data structures and methods, to assist the requirements engineer in ensuring the internal consistency of the captured specification.

Most importantly, TranZit contains a *novel transformation engine*, which is capable of transforming a captured Z specification into an executable representation based on extensions to LISP, suitable for direct execution in an animation environment. This process is supported by an eclectic strategy combining automated transformation with user assistance, to overcome many of the well-documented problems associated with transforming non-executable clauses in formal specifications.

1. Introduction and Problem Definition

It is an accepted fact that as time progresses, more everyday products are incorporating some form of computer system into their basic design. As the cost of powerful computer hardware continues to fall, driven by recent advances in semiconductor technology, it is now possible for manufacturers to build ever more sophisticated embedded computer systems into their products, to meet the increasing consumer demand for features and information. Computer systems are trusted to control everything from washing machines to jet aircraft; from ATM machines to stock markets. Indeed, such is the proliferation of computer systems in the modern world that it is hard to conceive of many aspects of everyday life that are not influenced by computer technology.

The massive increase in the availability of affordable desktop PC's to both the business and home user, has opened up a huge global market for computer software, which now incorporates a massive range of applications. In particular the consumer demand for recreational software has created a highly lucrative computer games market, whilst revolutions in the communications and information technology industries has made such things as mobile phones, satellite television, teleworking and home connection to the Internet commonplace. Together, these aspects of computing are already having a marked effect upon the sociological factors that influence the way our civilisation is evolving.

While the software applications market continues to grow year on year, likewise a similar increase in the embedded computing marketplace has seen computer systems begin to control much more of the equipment and services we depend on in our day to day existence. In recent years computers have been trusted to control more *safety critical systems*, such as aircraft *fly-by-wire* control systems or Nuclear Power Plant control systems, in which the reliability and performance of the controlling software is paramount in safeguarding the well-being of human beings under its influence.

Our technological society has now evolved to the point where almost everyday, we place our physical, social and economic wellbeing in the hands of computer systems, and place

our trust in the fact that they will perform the tasks required of them flawlessly. Yet as these systems become ever more complex, and the performance demands placed upon them are forever increasing, how can we be sure that the computer software they embody will deal correctly with the chaotic myriad of events that constitute the real world?

It is this dichotomy which continues to perplex software developers throughout the world: How to meet the challenge of developing reliable and efficient computer software, which satisfies the needs of the customer, whilst maintaining control of project costs and timescales?

In this introductory chapter we will explore the background to this problem through;

- An investigation into the history and reasons for the evolution of Software Engineering,
- An analysis of the problems addressed and highlighted by various approaches and techniques adopted in Software Engineering,
- A discussion of the software development lifecycle and the costs which it embodies.
- Investigation of the Requirements Capture lifecycle phase, with a view to improving the quality of the deliverables produced.

The chapter aims to present a clear background to the problem domain and set the context for the work presented in the remainder of the thesis. In addition, this chapter will define the need for such work and identify clear plans and objectives for the subsequent project work.

1.1 The Risks and Costs of Developing Software

Whilst recent times have seen the cost of computer hardware continue to fall year on year, the costs associated with developing computer software have not. As long ago as 1977, Lehman (1980) suggested that software development costs in the USA exceeded \$50 billion a year, representing more than 3% of the American GNP. Accurate figures

representing actual costs today are difficult to establish, often due to the fact that development organisations regard this information as commercially sensitive and therefore do not publish figures openly. However, with the huge increase in the number and size of commercial software houses seen in recent years, it is likely that the equivalent costs today are several times this figure. Indeed Boehm (1987) suggested that in 1985, worldwide software costs exceeded \$140 billion, and would continue to grow at a rate of 12% per year. Projecting these figures into the future suggests that by the end of the century, software development costs will exceed \$600 billion a year.

This figure represents a huge investment by the software industry, and with so much money at stake the cost of failure may have devastating financial effects on organisations developing such computer systems. Yet the enormous global market for IT and computer products means that if computer systems can be made to work to specification and meet the customer need, then software development as an industry can return generous profits. One need look no further than the phenomenal success of the Microsoft Corporation to understand what the returns on such an investment may be.

However, even apparently successful organisations are not immune from the problems associated with complex software systems development: As reported by Forsberg (1997), during the Atlanta Olympics the IBM Corporation suffered a number of high-profile problems with their \$40 million Olympic Information Integration system. The result was that twelve News wire services that had contracted to the IBM system had trouble obtaining accurate competition results. The effect of this high profile failure not only had a negative impact on the credibility of the IBM organisation itself, but also had direct financial implications for the customers of the system. When questioned on the matter at a later date, the IBM project manager, Luis Estrada, conceded that the system had broken down “.. *because user requirements were not understood*”.

This statement has far reaching consequences, in that the apparent failure of a costly computer system was not attributed to poor design, poor implementation or even poor testing. It is purely the case that the system did not do what the users wanted it to do. One could be forgiven for thinking that it should be obvious that the system should do

what the users want it to do, and that one should not embark upon such a project until the system requirements are fully agreed with the customer. Yet as discussed below, the problem of *engineering requirements* for complex computer systems is perhaps one of the most challenging fields of computer research today.

1.1.1 Developers Under Pressure

At the same time as the costs involved in major software projects continue to increase, so the software developer is under increasing pressure from two opposing forces. On the one hand there is competitive pressure from management and the business to complete developments faster, increase productivity, get the product to market sooner and make it cheaper than before. On the other hand there is pressure from increasingly technologically minded customers for high quality IT solutions which add value to their business.

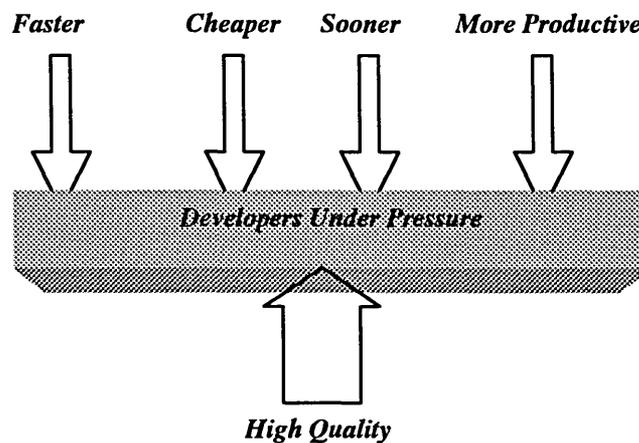


Figure 1-1: Developers under Pressure

In order to meet these objectives, the development team must focus their efforts on achieving the right product first time. Hence any methodology we choose to develop software must be much more than simply a proforma for writing software. It must be an integrated process for engineering the right product to meet business objectives, costs and timescales, whilst at the same time satisfying customer requirements and quality measures.

1.1.2 The Need to Capture What the User Requires

From the preceding discussion, an immediate conclusion one could draw might be that projects are more likely to meet budgets and generate customer satisfaction, if more time were spent analysing the users' requirements during the study phase of the project. This view is now generally accepted and is supported by data from the NASA organisation shown in Figure 1-2 (reproduced from Forsberg, 1997), which relates to several of their space system projects. The diagram indicates the benefits of conducting an effective project study phase as a percentage of the development budget overrun costs (source: W. Gruhl NASA-HQ).

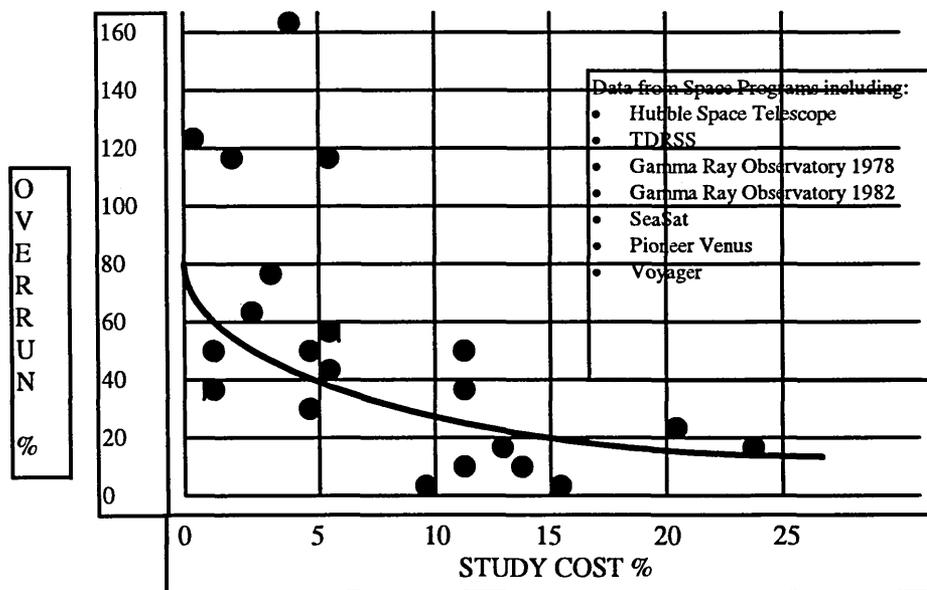


Figure 1-2: Study Phase Costs as a Percentage of Development Costs

This data highlights the view that if no requirements study is started before development begins, then project overrun costs can be up to twice the original project budget estimate.

This was aptly demonstrated by the Geostationary Operational Environment Satellite project, which was initiated by NASA in 1985. This project was crucial to the replacement of the existing weather satellites, which were expected to reach the end of their operational life by the end of the 1980's. The project was planned to be completed,

ready for a first launch, in 1989 (a total of 4 years) and had a budget in excess of \$500 million. A crucial decision made in the evolution of the project was that management decided to skip the traditional requirements study phase. The rationale for this decision was that it was considered that the project amounted to a *replacement* exercise for existing satellites and therefore the operational characteristics of the required system were well understood. This decision proved to be a grave mistake, with the result that the first satellite was launched five years late, by which time the project costs had soared to over three times the original estimate. The resultant political outcry is well documented by Kuznik (1994) in his article “*Blundersat*”, and almost caused the project to be completely cancelled.

It should be clear from the previous discussion that any complex engineering problem needs careful analysis at project inception in order to reduce the risk of budget overrun and possible system failure. Yet with so much more emphasis being placed on software solutions, how have software developers responded to this challenge? Most importantly, what causative problem agents have been identified in the software development process, and what thinking has evolved to mitigate these problems? To find the answers to these questions, it is necessary to examine the evolution of Software Engineering itself.

1.2 Software Engineering Processes and Practice

In the early years of commercial software development, a highly unstructured, *ad-hoc* approach was taken in addressing the development process. However, as development costs began to rise, the companies involved soon began to realise that software development required much more than simple programming. In effect it became necessary to consider the *environment* into which the software would be delivered, including psychological, organisational, ergonomic, economic and performance factors. It was realised that computer systems needed to be *engineered* to fit into organisations and meet their business objectives, rather than simply be *programmed* to provide specific functionality. The needs of the *user*, rather than the needs of the program were beginning to become important.

The 1960's saw a huge increase in the computing power available to the programmer. This made it possible to apply computer-based solutions to a whole range of business processes hitherto thought to be impossible or too expensive to implement.

Computers could now be used to automate payroll systems, manage stock control, produce management reports, and perform many other traditionally manual clerical tasks, as the amount of data that could be stored and manipulated within the system increased dramatically. However, as the systems programmers tried to apply their old, *ad-hoc* development techniques to these much larger problems, it became apparent that simply scaling up the *programming effort* failed to produce acceptable solutions. Many systems under development began to be delivered late, showed spiralling costs, were difficult to maintain or modify and most importantly failed to meet the expectations of the users.

The software industry was in a state of *crisis* (Naur and Randell, 1969), and as a result it was recognised that the application of *generic engineering principles* to the development of software was the only way for the industry to retain its credibility.

Many solutions were postulated, however the essential conclusion reached was that the problems associated with developing large, complex software *systems* were fundamentally different to those involved in developing an isolated computer *program*.

One of the major differences subsequently identified highlighted the fact that deficiencies in the *analysis of requirements* for large software systems made a significant contribution to the overall development problem. Indeed it was stated by Alford and Lawson (1979) as far back as 1979 that “*In nearly every software project which fails to meet performance and cost goals, requirements inadequacies play a major and expensive role in project failure*”.

However, what aspects of the generic engineering principles outlined, were needed to transform software development from an art into an engineering discipline?

1.2.1 Software Engineering Practice

Although coined in the late 1960's, the term *Software Engineering* has evolved in the intervening years to cover a range of software development issues. There are many proposed definitions of the term Software Engineering, although Sommerville (1985) provides one of the more concise when he states:

“The practise of Software Engineering is concerned with building large and complex software systems in a cost effective way.”

The use of the term *Engineering* implies much more than simple programming, involving a thorough analysis of the problem, a systematic approach to design, and a rigorous validation method, all supported by effective documentation.

The use of an engineering approach was intended to break the traditional cycle of the software system relying for its development and maintenance, on a small group of programmers who knew the system internals in intimate detail. It soon became clear that to surmount the problems involved in developing and maintaining such systems, more than one role was needed in the software engineering process. In particular, there was early recognition that more work was needed in the analysis and design phases and this in turn led to specific roles being generated, including the *Systems Analyst* or *Analyst Programmer*.

As these roles developed, so it became necessary to formalise the interface between them and hence discrete phases of system development began to evolve, associated with the particular specialisations. Similarly, the need for a systematic approach to software engineering naturally led to the evolution of so called *methodologies*, designed to standardise and organise the information presented at each phase of development.

1.2.2 The Software Lifecycle

The software lifecycle is an extremely important model in defining the nature of software systems development. Not only does it clarify understanding of the associated problems, but it also defines *an engineering process for software development*. This process

consists of several discrete engineering phases, each of which is associated with a set of deliverables and objectives.

It is important to recognise that there is no single definitive view of the information embodied within the software lifecycle model. Rather the discrete phases identified must be organised according to various factors, possibly including the nature of the application being developed, the organisation and management of the development team, project timescales, cost targets and deliverables.

According to Avison and Fitzgerald (1988) the software lifecycle model grew out of three main observations:

- Firstly, a growing appreciation of the importance of the systems analysis and systems design roles within the development process, as well as the systems implementers.
- Secondly, the realisation that the rate at which organisations were growing necessitated the need to develop integrated and maintainable *information systems*, rather than simple one-off programs.
- Finally, the desirability of a standardised approach or *methodology* for the development of software systems.

The notion of a methodology recognises that software systems pass through a number of discrete phases in their development and use. One of the earliest methodologies was that developed by the National Computing Centre (NCC) in the United Kingdom and is described by Daniels and Yeates (1971), and later by Lee (1979). The methodology, which came to be known as *conventional systems analysis*, is based on the so-called *Waterfall model* of software development, proposed by Royce (1970).

In essence, conventional systems analysis methodology embodies the following discrete phases:

- *The Feasibility Study Phase* examines current problems and proposes alternative solutions, based on simple cost/benefit analysis.

- *The System Investigation Phase* follows the feasibility phase and the developers then proceed to assess the system requirements, record constraints and problems with current working methods.
- *The System Analysis Phase* analyses the current system with view to addressing why such problems exist, where automation will help, and to define the boundary of the proposed system.
- *The System Design Phase* involves the design of both the computerised and manual parts of the system.
- *The Implementation Phase* concerns the implementation and testing of the computerised parts of the system. The manual parts are documented and users are trained. Master files are set up and the system goes through a period of trial running before being brought into service.
- *The Review and Maintenance Phase* is the final stage of the methodology and addresses any changes required to the system to ensure efficient running, and also review the performance of the system against the original requirements and objectives.

From the description above, it is clear that the original NCC methodology was (rightly) designed around the state of computer technology as it existed at the time. In more recent years, the development of new techniques for software development has offered alternative approaches based on emerging wisdom and experience. In particular, the following techniques represent some of the major developments within the evolution of software engineering:

- *Stepwise Refinement* (Wirth, 1971),
- *Jackson Structured Programming* (Jackson, 1975),
- *Structured Systems Analysis and Design* (Weinburg 1978, Yourdon and Constantine 1979, Gane and Sarson 1979 and DeMarco 1978) , and
- *Object Oriented Analysis and Design* (OOA/OOD) (Booch 1994, Shlaer and Mellor 1992)

It is undoubtedly the formalisation of these methods that has led to the acceptance of software engineering as a true engineering discipline.

Whilst these methods differ in syntax and semantics, they all essentially encompass the same prescriptive approach to the development process. Hence, today it is generally accepted that the software lifecycle model encompasses the following familiar discrete phases:

- *Requirements Capture*. This phase is concerned with *what* is to be designed rather than *how* it is to be designed. The system features, performance constraints and operating environment are established by detailed discussion with the users. Once these have been elicited, they must be captured in a complete, concise and unambiguous fashion to form a *specification* of what is to be designed. Again, due to the importance and complexity of this phase, it has evolved into an area of study in its own right. This has come to be known as *Requirements Engineering*, and is of primary interest in the remainder of this thesis.
- *Software Systems Design*. This phase seeks to describe a number of software elements whose characteristics can be implemented in the target programming language.
- *Implementation*. In this phase the software design is implemented in the target programming language. The result is to produce a number of programmed components or *modules*, each of which implements a specific part of the software design.
- *Unit, System and Acceptance Testing*. In *unit testing*, the implementers verify that each of the modules implements its software design correctly. This is generally followed by *system testing*, in which all the individual modules are *integrated* to produce a complete software system. This completed system is then tested to ensure that it meets the original requirements of the system. In *acceptance testing*, the entire system is installed and brought into partial service, normally for a trial period, to allow the users to observe the system operating *in situ*. This provides further confirmation that the system requirements have been met.
- *Operation and Maintenance*. Once the user has accepted that the system is fit for purpose, it will be brought into full operation. It is often the case that throughout this period (i) problems with the system operation will occur which were not identified in the testing phase, or (ii) users' requirements will change as their business evolves. If it

is to remain useful, the software system must be *maintained* to improve, enhance or correct features of the system.

These phases are normally represented using the general *waterfall model* of software systems development as shown in Figure 1-3:

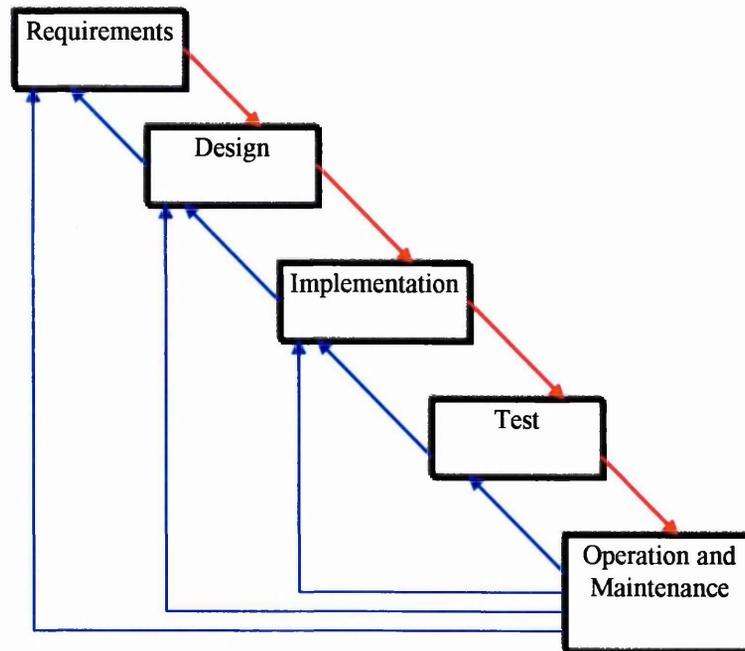


Figure 1-3: Waterfall Model of Software Systems Development

It is important to recognise that each transition on the software lifecycle model is associated with the cumulative cost of completing the previous phases. The model apparently shows a seamless flow of information from one stage to the next, starting from a statement of requirements through to delivery and maintenance of the final system. Using the model, one should therefore be able to predict with a fair degree of accuracy the overall cost of system development. Yet whilst this model is suitable as a skeletal outline of the development process, it fails to capture the dynamic behaviour of the system under development in any real sense. If development were as straightforward as is implied by the software lifecycle model alone, then it is difficult to perceive of any great problems with controlling the associated cost. However, since it has been demonstrated that this is not the case, even in mature organisations, there is clearly hidden cost in this model that needs to be identified.

1.2.3 Identifying Costs in the Waterfall Model

An understanding of why software development costs so much more than anticipated, is key to improving the overall development process. The goal is to establish which of the lifecycle phases is responsible for the apparently *hidden costs*, which in turn take the development over budget for some reason. It is then possible to set about improving the way in which particular lifecycle phases are managed, to reduce the cost overhead.

The software lifecycle model is interesting from the viewpoint that information is seen flowing back to previous phases, implying an iterative or *cyclic* approach to development. Apparently, this also includes the requirements capture phase. The implication here is that we may need to revisit certain phases of the lifecycle several times before we can complete a particular transition to the next phase. Hence the first glimpse of hidden cost is the fact that the cost of a particular transition may not simply be the cumulative cost of the previous forward transitions. It may appear obvious, but many development organisations still make this simplistic mistake when costing projects.

More importantly, it can be seen that changes made during the Operation and Maintenance phase of the lifecycle may have a direct effect on *any* of the previous phases of the system lifecycle, and as such may have a high cost impact.

This then is the essential problem with developments based purely on the waterfall model as highlighted by McCracken and Jackson (1982) and also by Gladden (1982). The model assumes that users are capable of stating their requirements at the outset of the project, and that these requirements will be static during the lifetime of the system. However as Brooks (1987) points out, users often do not know what their exact requirements are, and even if they can be determined at some point in time, they are likely to change.

In order to generate an improved version of the software development lifecycle, it is therefore necessary to consider the dynamic behaviour of software systems and their environments over time in order to uncover the true sources of development cost. This process is termed *software evolution*.

1.2.4 The Implications of Software Evolution for Requirements Engineering

Software evolution (or sometimes Software Maintenance) is the process of changing the delivered system, either to correct errors introduced at some phase in the software development or to address the *evolving needs* of the environment into which the system was originally delivered.

Lehman (1980), has suggested that there are five rules which govern the evolution of software systems:

- *Continuing Change*. A program used in a real world environment must change or become less useful in that environment
- *Increasing Complexity*. As an evolving program changes its structure becomes more complex unless efforts are actively made to avoid this phenomenon.
- *Program Evolution*. The process of evolution is self-regulating, and measurement of attributes associated with different releases of software show statistically significant trends and invariances.
- *Conservation of Organisational Stability*. Over the lifetime of a program, the rate of development of the program is approximately constant, and independent of the resources devoted to system development.
- *Conservation of Familiarity*. Over the lifetime of a system, the incremental system change in each release is approximately constant,

As Sommerville (1985) points out, these laws are not universally accepted, however it is certainly the case that the first two laws are probably applicable to every large software system which has ever been developed. However, the important point is that there is an assertion that any software system *will change*, either during its development or after it has been installed. If this is the case then what are the costs associated with implementing these changes?

It has already been noted that each phase in the Software Development Lifecycle carries an associated cost to the developer. In general this cost is dependent upon the nature and

complexity of the particular system being developed, however Boehm (1981) has suggested figures for general *types* of system based on experimental observations. From the work of Boehm, the inference is that during the system maintenance phase, the further it is required to go back in the software lifecycle to implement a given change, the higher the cost involved. The reason being that maintenance work re-incurs the cumulative cost of all the intervening lifecycle phases.

For example, consider an error made in the implementation phase that is discovered in the testing phase. The cost associated with rectifying this problem is the cost of re-coding and re-testing the incorrect part of the software. However, if the system is delivered to the user and a feature of the system does not satisfy the *requirements* of the customer, then the cost of rectification is much higher. In this case, it is the cost of re-specifying the requirements, and subsequent re-design, re-code and re-test of the various components affected by the change.

Boehm's research has shown, and it is now an accepted view, that the costs associated with software maintenance are extremely high and in some cases may exceed the original development costs of the system by a factor of between two and four times. It is also an accepted view that the majority of maintenance costs originate not from the need to correct design and implementation errors in the system, *but due to changes to the system requirements*.

According to the first law of software evolution, we must accept that the system requirements will change in the long term. However, in order to reduce the cost of this effect for as long as possible, *it is paramount that every effort is made to establish an accurate representation of the users requirements during the initial requirements capture phase of the software lifecycle*.

We must be clear the Software Engineering process begins with the *needs of the customer*. This involves capturing *what needs to be designed* at the outset of the software lifecycle, rather than *how it is to be designed*. The next task is to explore ways in which we can improve the lifecycle model to achieve this.

1.2.5 Improving the Lifecycle Model

If we are to control the costs associated with the development process, then it is clear that we must improve our model of the way in which software is developed, and in particular we must focus on the Requirements capture phase as a key element of achieving lower cost.

Several ideas have been postulated, perhaps the most pragmatic of which is Boehm's Spiral Model (Boehm, 1988), which shifts the emphasis from achievement of development milestones (as in the Waterfall model), to the analysis of the risks (and costs) of continued development. Each phase of development is depicted as a loop in a spiral with the radial co-ordinate representing the actual costs incurred so far. A phase completes at each transition of the X-axis, at which point a decision is taken on how to proceed based on an evaluation of objectives, determined constraints, development alternatives and risk analysis.

As Dorfman (1997) points out, the Spiral model advocates no particular approach to dealing with each development phase, and makes explicit the idea that the form of a development cannot be precisely determined at the outset. In this way, the completion of each spiral loop gives an opportunity to re-evaluate the development from a number of perspectives, including changes in user perception, changes in technology and ongoing financial considerations.

The main advantage of the Spiral model is that it is not prescriptive and is applicable to a wide variety of project pre-conditions. For example, if it is the case that the user requirements are well understood at project inception and that the associated development risks are low, then the Spiral model effectively collapses to a standard Waterfall model. However, in the case where the requirements are less certain, other development models can be derived from the basic Spiral model, which better reflect the needs of the particular issues raised.

1.2.6 The Case for Prototyping

Whilst the spiral model is a more pragmatic approach to the development of real-world software systems, it does not in itself improve the process of requirements capture. What is required is a more dynamic approach to the identification of user requirements.

To address this problem, a further extension to the standard waterfall model, is the *Prototyping model*, which Gomaa and Scott (1981) and others advocate as a good approach to support the requirements analysis phase within the development model.

The term *prototype* has different meaning to different people, however the definition adopted here is that of Henderson and Minkowitz (1985) who assert that a prototype is; "*.. a skeletal, inefficient, throw-away implementation of the precise functionality of the eventual system.*".

In this model, the implication is that some form of skeletal system capability is constructed with a minimum of formality, which can be demonstrated to, and experimented with by the user. The prototyping lifecycle model is shown in Figure 1-4:

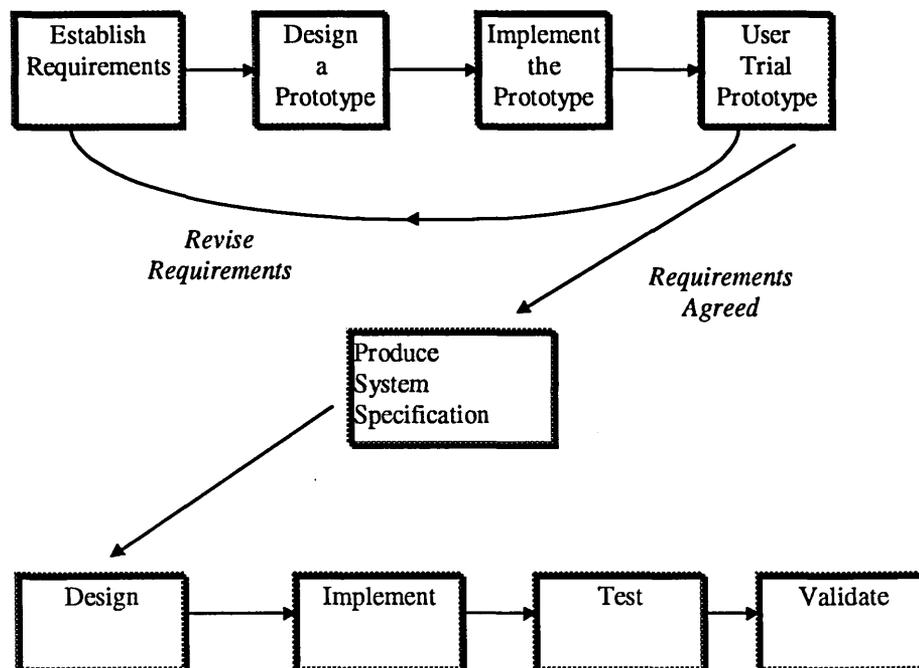


Figure 1-4: Prototyping Lifecycle Model

It is seen that the first phase of this model involves an iterative cycle of requirements analysis, modelling, implementation and then trials against the users' perception of what the system should do. This cycle continues until the user agrees that the prototype system captures the requirements of what the system needs to do. The requirements are then documented as a system specification, which forms the contractual agreement upon which development of the product will be based. The development of the product then continues according to the standard approach described previously.

The idea behind prototyping stems from the recognition that prototyping enhances the *communication* between customer and designer about the proposed system. Indeed as highlighted by Alavi (1984), many developers have commented "*The end-users are extremely good at criticising an existing system, but not too good at articulating or anticipating their needs*".

As stated by Wassermann and Shewmake (1985), a prototype system allows users to experiment with their ideas, which may even cause them to change their views about what they want the system to do. More importantly, observing this prototype in operation, the user is much more likely to notice inadequacies in the specification.

The ultimate aim of prototyping is therefore to improve the quality of the specification of requirements, which in turn may shorten the development time and reduce unnecessary costs due to re-work.

As indicated in the IEEE Recommended Practise for Software Requirements Specifications (IEEE, 1993), many tools and methods have been developed to support the idea of prototyping, assisted by the evolution of so called Fourth Generation languages (4GL's), many of which have powerful, abstract semantics. However, in recent years researchers have looked to the possibilities of generating a prototype directly from the system specification. Such research has led to the idea of *rapid prototyping* and the development of *animation systems* and *executable specifications*.

1.2.7 Rapid Prototyping, Animation and Executable Specifications

Whilst the reasoning behind prototyping is sound, the development of a prototype during the requirements capture phase may itself add an unacceptable cost to the overall project development. This is especially the case if the prototype will be discarded once requirements capture is complete. An extension of this idea is therefore the generation of a *rapid prototype* (Gomma, 1997).

By the term *rapid prototype*, we imply a prototype that can be generated on demand by *automated tools*. This is especially useful in the case where several cycles of requirements elicitation are anticipated, as there is unlikely to be enough time in any non-trivial project to develop several prototypes by hand. Therefore, the major advantage of rapid prototyping by automated tools is that, in addition to the prototyping benefits outlined previously, the increased efficiency offered facilitates *iterative refinement* (Hartson and Smith, 1991).

Because automated tools allow rapid representation of ideas, it is possible to experiment quickly with differing scenarios, increasing the possibilities for iterating to a final specification that meets the user's needs. The process of producing a rapid prototype from a specification and then experimenting with it in an interactive fashion is often termed *animation* (Knott and Krause, 1988).

The rapid evolution of fourth generation programming languages, AI techniques and application generators has added considerably to the possibilities for *computer-based rapid prototyping*, due to the declarative semantics they embody. However, more recently, researchers have focused on the possibilities for combining computer-based rapid prototyping techniques with *formal specification* as an aid to validating complex specifications against user requirements.

Since formal specifications embody mathematical semantics, this is seen as an ideal mechanism for improving the *quality* of the specification by eliminating ambiguity, increasing precision and completeness, as well as providing a foundation for validating

the requirements embodied. The merits of this approach are well argued by Dick *et al.* (1990) as follows:

" Rapid Prototyping is seen as an important method of validating a specification against its informally perceived requirements. This is especially so when the specification language used is not easily understood by the non-specialist reader who, nevertheless, has a strong interest in the consequences of the specification."

The key issue here is that whilst formal specification languages may bring several advantages to the requirement engineering task in improving the quality of the specification, these languages are not easily understood by the non-specialist customer. However, the specification is a *contract* between the customer and the developer as to what is to be developed. If the customer cannot understand the specification, how can the customer agree that it is a true representation of the system requirements?

1.2.8 The Application of CASE Tools

From the preceding discussion, it is clear that requirements engineering based on the use of formal methods presents the specifier with several challenges. Not only must the specifier capture the system specification in a formal notation and be able to manipulate it efficiently as requirements acquisition proceeds, but the need to *validate* that the specification is a true representation of the system requirements is also an important goal.

To address these problems in the most effective way a computer-based tool is required to assist the requirements engineer in constructing, manipulating and maintaining the internal consistency of a formal specification during the requirements engineering task. In addition the tool should support the generation of a rapid prototype of the system directly from the formal specification. This rapid prototype may then be loaded into an animation environment and exercised to demonstrate the expected behaviour of the system, or discover new properties of the system represented by the specification. This animation process is referred to as validation of requirements by execution.

However as Dick *et al.* continue;

“By their very nature specification languages are non-algorithmic, which inhibits direct execution. Rapid prototyping, therefore, involves making an interpretation of the specification in some language that does lend itself to direct execution.”

The key point here is that, in general, specification languages are non-executable (Hayes and Jones, 1989). Hence, it is necessary to *transform* the formal specification in some way, in order to produce an equivalent executable representation suitable for use as a rapid prototype. For a non-trivial specification, this transformation process may be extremely complex, which generally prohibits the generation of a *consistent* and *accurate* executable representation by hand. As indicated, it is also the case that certain elements of most formal specification languages do not have a directly executable representation, requiring the development of a strategy that can be embodied in a computer-based tool to highlight such problems to the requirements engineer.

With this in mind, the main purpose of the research project described in this thesis is the development of a computer-based tool to support the use of formal methods in requirements engineering.

1.3 The Objectives of the Research Programme

This project concerns the support of requirements engineering and the use of formal methods by computer-based tools. It has been identified that there is a need to ensure that a complete, unambiguous statement of requirements is developed before system design begins, in order to minimise project costs and produce the *right* product. It has been suggested that use of a formal specification language may help to achieve this objective, however a more dynamic approach is needed to ensure that the customers can understand and validate that the specification is indeed a true representation of the required system.

It is proposed that an executable representation of the system specification in the form of a rapid prototype, may be used to allow users to *validate the specification by execution* in an animation environment. However, in order to assist the requirements engineer in producing a rapid prototype, computer-based tools are required to maximise the efficiency and to ensure the consistency and correctness of the transformation process involved.

Therefore, the objective of this research project is to **research, implement and critically evaluate a CASE tool for requirements engineering, which will allow the capture of formal specifications and subsequently produce an executable representation of the specification, suitable for use as a rapid prototype within an animation system. The project should make a contribution to knowledge in terms of addressing the problems association with the transformation of non-executable specifications.**

This is to be achieved by a number of objectives as outlined below:

- Definition of a *process model* for requirements engineering, which integrates the proposed toolset.
- Definition and implementation of a computer-based tool which can be used to *capture and store* specifications efficiently in a formal notation.
- Definition and implementation of an *analysis* system for checking the *internal consistency and correctness* of the specification which is captured.
- Definition and implementation of a computer-based mechanism to automate (as far as is possible) the *transformation* of the captured specification into a *procedural or executable representation*, suitable for use as a rapid prototype in an *animation system* for the purposes of *validating* the captured specification by execution.
- Testing and evaluation of what has been achieved including comparison with other computer-based requirements engineering tools, and demonstration of the efficacy of the system developed through practical application in an animation environment.

The tool that has been developed to meet these objectives is known as *TranZit*. It is the research and development of the *TranZit* tool that forms the basis of the work presented

in this thesis. *TranZit* is designed to integrate with an animation environment termed *ZAL*, which has been developed as part of a parallel research programme. Together these tools form the key components of an integrated requirements engineering environment termed the *REALiZE Toolset*.

1.3.1 The Project Plan

The achievement of the project objectives is dictated by the following work plan, which was agreed with the project supervisors at the outset of the project.

- 1) *Review the current literature to gain knowledge in the field of Requirements Engineering.*
- 2) *Review current Requirements Engineering Toolsets, with a view to determining the state-of-the-art, and the advantages that the TranZit tool can bring to this field of research.*
- 3) *Research and develop a Requirements Engineering Process which will aid in the process of specification verification, upon which to base the toolset.*
- 4) *Research and develop the Windows-based TranZit editor and specification capture system based on the Z notation.*
- 5) *Sub-system Test, Review and Refine.*
- 6) *Research and develop the TranZit Analyser Subsystem (TAS) involving an integrated syntax and type checker for the Z notation.*
- 7) *Sub-system Test, Review and Refine.*
- 8) *Release the TranZit system for user acceptance testing and perform early-life monitoring.*
- 9) *Review the possibilities for automated transformation of the Z notation into a procedural representation. Liaise with the development of the ZAL language to define an executable subset of Z, and associated ZAL grammar.*
- 10) *Research and develop the TranZit Transformation Engine (TTE), with the objective of automating the transformation of a captured Z notation directly to an executable representation in the ZAL language.*
- 11) *Sub-system Test, Review and Refine.*
- 12) *Integration test TranZit with the ZAL animation environment*
- 13) *Generate results by exposing the system for user acceptance testing.*
- 14) *Review, define possibilities for future work and draw conclusions.*

1.4 Thesis Structure

The remainder of this thesis reports on the research, development and critical evaluation of the TranZit tool.

Chapter two concentrates on putting in place the required background knowledge concerning requirements engineering and formal methods, in particular the Z notation.

In Chapter three, a taxonomy of requirements engineering tools is developed, which leads into the development of a set of *product* requirements for an automated tool, based on analysis of the activities within the requirements engineering task. The chapter concludes by defining a requirements engineering *validation process* into which the TranZit tool is to be integrated. This process is central to developing a foundation for using the TranZit tool within an integrated animation environment, in order to solve real-world problems.

In Chapter 4 the realisation of the TranZit tool is explored with a view to understanding the design and development of the TranZit editor and analyser subsystem (TAS).

In Chapter 5, the realisation process continues with a discussion of the design and development of the TranZit transformation system, and in particular the approach taken by TranZit in addressing the transformation of non-executable clauses in the Z notation.

In Chapter 6, the testing and evaluation of the TranZit system are discussed. This chapter begins by identifying the software test strategies employed to ensure the *quality* of the system, before quantifying the *usability* of the system as perceived by the users. A comparative evaluation of TranZit with other requirements engineering tools is then presented in order to identify the contribution made by the TranZit tool to the requirements engineering task. Finally, the chapter concludes by describing two detailed case studies, which highlight the use of TranZit as an integrated component of a practical animation environment.

Finally, in chapter 7 general results and conclusions are discussed which identify the achievements of the research programme against the original project objectives, the possibilities for future research and general conclusions concerning the application of animation and formal methods in requirements engineering.

2. Requirements Engineering and Formal Specification

Many systems design methodologies have been developed in recent years to assist in structuring the tasks involved in software engineering, by identifying the processes and information flows involved at each phase. However, these methodologies do not really address the issue that systems continue to be delivered late, run over budget and fail to meet customer requirements. Indeed, in his paper, Robinson (1994) states that as many as 50% of information system projects may still be considered to be failures due to inadequate specification of requirements.

Whilst it is widely accepted that all requirements should be fully defined and agreed before design begins, what are mechanisms required to achieve this? Indeed, what are the goals of the requirements analysis phase, and how do we know when the phase has been successfully completed?

To answer these questions, this chapter explores the background to requirements engineering, formal methods and specification validation, in order to understand the foundation upon which the subsequent project work is based.

2.1 What is a Requirement?

The first question to consider is what is a requirement? IEEE standard 610-12 (Dorfman and Thayer, 1990) defines a requirement as:

1. *A condition or capacity needed by a user to solve a problem or achieve an objective*
2. *A condition or capability which must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.*
3. *A documented representation of a condition or capability as in 1 or 2.*

Pohl (1993) in his paper 'The Three Dimensions of Requirements Engineering' provides a good definition of requirements engineering as:

“Requirements Engineering can be defined as the systematic process of developing requirements through an iterative co-operative process of analysing the problem, documenting the resulting observations in a variety of presentation formats, and checking the accuracy of the understanding gained.”

As Macaulay (1996) points out, Pohl’s definition is important because each part of the definition leads to a number of questions:

“... the systematic process of developing requirements ...”

- How can we define a systematic process when there are so many unknown factors at the beginning?

“... through an iterative, co-operative process of analysing the problem ...”

- How do we know when analysis is complete and all the requirements have been gathered?
- The term co-operative refers to co-operation between people. Who should be involved in the process? How will they communicate with each other? How will they reach agreement on the process?

“... documenting the resulting observations in a variety of representation formats ...”

- What representation formats should be used and how should the results be documented?
- What standards and which notations should be adopted?

“... checking the accuracy of the understanding gained ...”

- How will we measure the accuracy of understanding and hence know when the checking process is finished?

- Will everyone involved in the requirements engineering process have the same understanding?

These are important issues to understand and must be addressed by any method developed to assist in the requirements engineering task.

2.1.1 Requirements Engineering Methods

Dorfman (1997) suggests that Requirements Engineering methods may be roughly divided into four basic categories as shown in Figure 2-1:

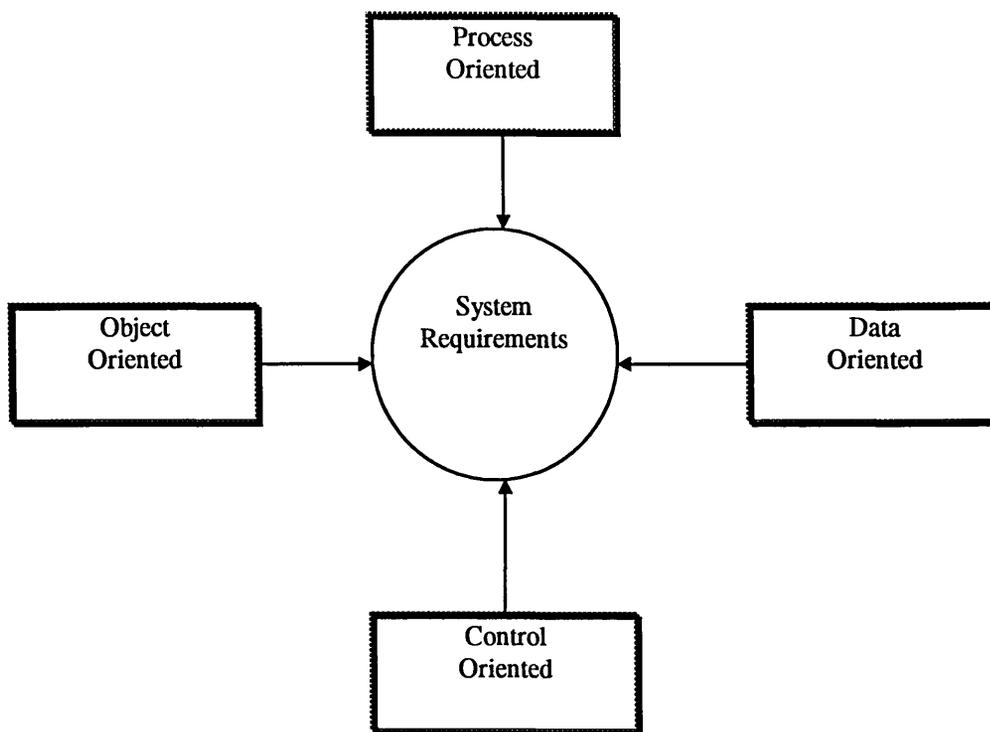


Figure 2-1: Categories of Requirements Engineering Methods

The methods are described as follows:

- *Process Oriented*: The primary view of the system is that it takes some form of input and produces some form of output as a consequence. The requirements engineering task focuses on identifying these *transformations*. Structured Analysis (Ross and Schoman, 1977), and model-based formal specification methods using VDM (Jones, 1990) and Z (Spivey, 1992) are good examples in this category.

- *Data Oriented.* The system is viewed as consisting of state-dependent data structures. The requirements engineering task focuses on identifying the data components which make up the system. Entity-Relationship Modelling (Reilly, 1997) and JSD (Jackson, 1983) are good examples in this category.
- *Control Oriented.* The requirements engineering task focuses on identifying the system control mechanisms, such as process synchronisation, activation and concurrency. Such considerations are important in the specification of real-time control systems such as those required for avionics applications. Methods such as the real-time extensions to SA (Ward and Mellor, 1985) are important examples.
- *Object Oriented.* The requirements engineering task focuses on the classes of objects which constitute the system and the relationships between them. Formal methods approaches using VDM++, Z++ and languages such as object-Z are good examples. These are well documented by Lano (1995).

As Dorfman himself points out, this categorisation should not be taken as absolute. Rather, most requirements engineering methods take ideas from all categories, but one view is mainly paramount.

However, in common with all these approaches, the requirements engineering task can be considered as the process of constructing a *model* of the problem domain, and populating this model with functional, organisational, social and economic factors elicited from the environment into which the final system is to be delivered. Hence the modelling process needs to consider much more than the simple technological aspects of the problem.

Again, the IEEE (1984) guidelines make explicit the difference between the model upon which the *specification* is based, and the model upon which the *application software* is built, which is likely to be purely technologically based. Hence the model chosen for the requirements engineering process must possess specialist characteristics which enable it to capture the diversity of information presented during requirements elicitation. According to Verheijen and Van Bekkum (1982) these characteristics should include:

- *A high level of abstraction*: allowing the users' view of the system concepts to be captured directly.
- *Human Readability*: The language in which the model is presented will be used for validating the contents of the specification with the user. User understanding of the specification is therefore of prime concern.
- *Precision*: The language in which the model is presented must be unambiguous and ideally allow formal checking of consistency.
- *Specification Completeness*: The model language must be flexible enough to capture all aspects of the specification, not simply functional aspects.
- *Mapping to later Development Phases*: The model will be an input to the analysis and design phases to follow. There should therefore be an efficient mapping to the methodologies and procedures to be adopted in these lifecycle phases.

Yet how is the information required to build the model of requirements to be gathered, and what activities constitute the requirements engineering task?

2.1.2 The Activities within Requirement Engineering

In terms of activity, Davis (1993) describes the two main events that occur during the requirements engineering process as *problem analysis* and *product description*.

Problem analysis involves *information gathering* using techniques such as interviewing, observation and questionnaires in order to understand the problem domain from the viewpoints of the users and the constraints of the current system (if one exists). *Product Description* is the general process of correlating and organising this information into a description of the expected external behaviour of the product. It is recognised that these processes are not completely independent and that there may be several iterations involved in each phase

Most importantly, the requirements engineering process must consider the complete environment into which the system is being delivered, not simply technological issues. It

therefore follows that whatever *process* is to be used to capture such requirements, it too must be based specifically on the particular environment to be analysed.

This is especially relevant where a system is to be developed to meet a particular organisational need (e.g. a payroll system), as the requirements will be heavily influenced by the constraints of the work practises employed by the user company.

2.1.3 Factors Affecting the Requirement Engineering Task

The requirements engineering task is also heavily influenced by the *motivation* for introducing the new system.

For example, the requirements engineering task involved in developing a specification for a new, innovative product in the marketplace is likely to be very different to that employed to define the requirements for modifications to a legacy system already in widespread use.

In the first case, likely techniques used might include customer interviews, surveys and group meetings, to try to elicit as many different views on requirements as possible. These views will then be sorted, prioritised and amalgamated by the developing organisation, to produce a specification of requirements for a product which *meets as many of the requirements as possible within cost and time constraints*. This means that a view is taken on the merits of each requirement, which are then prioritised according to the *value* that they add to the final product. Thus in this case, the requirements engineering task inherently includes the idea that *not all requirements will be met by the final system*.

In the case of a legacy system, rather than be as open-minded as possible, the initial approach might be to interview everyone who works with specific components of the current system in order to build a model from which to develop new requirements. This model then forms the foundation of the improvements required to the system. The difference is that in this case, the user is likely to have a much clearer idea of what

benefits are expected to be achieved, and the requirements engineering task must attempt to address all the users' concerns with the present system.

2.2 A General Approach to Requirements Engineering

Some progress has been made towards *generalised* processes and methods for the requirements engineering task. Good examples are the RAISE project (George and Prehn, 1992), the RACE study (Bustard, 1994), the Cleanroom methodology (Mills *et al.*, 1987) and *meta-methods* for generic requirements engineering process models (Rolland and Plihon, 1996). However, it is generally accepted that it is very difficult to define a process that is effective in all situations. This is based on the problem that the process which is undertaken by the requirements engineer will be dictated by a number of variables including:

- The form or emphasis of the original project inception (e.g. whether the requirements are for a market driven commercial product, a technology driven project, a service driven project aimed at the needs of people or organisations, or a project where the quality of the system dictates *the physical safety or well-being* of the users)
- The type of system required (whether it is an extension to an existing system, a new system or a replacement system)
- The organisational, logistical and management ethos of the users' environment (e.g. the level of emphasis placed on budgets, resources, working conditions, staff morale, the presence of an open or closed management hierarchy and the willingness to change in the organisation)

These factors have established the view that the requirements engineering task is *situation-oriented*, in that the techniques and processes adopted must be tailored to the particular environment into which the system is to be placed. Since there are a huge diversity of problem situations, this in itself is one of the reasons why the task is seen as a difficult to define in generic terms.

However, whilst it is difficult to define a generic *process* which fits all problem domains, it is possible to define a *framework* of activities in the requirements engineering task as shown in Figure 2-2:

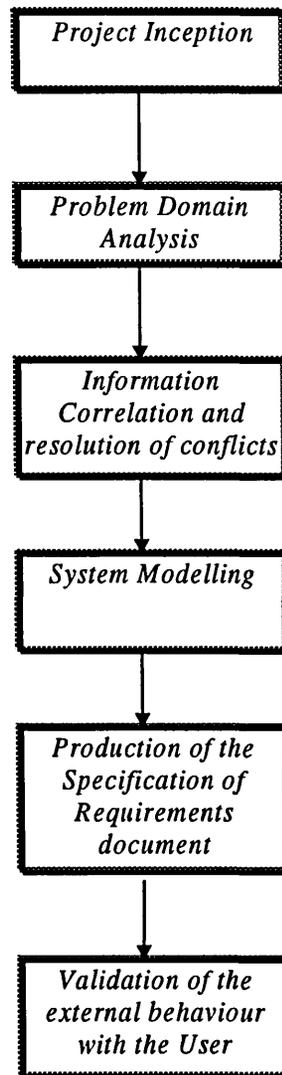


Figure 2-2: Requirements Engineering Process Framework

As pointed out by Macaulay (1996), there are good reasons to want to define a general approach to the requirements engineering task as highlighted by:

- The need to control projects and produce standardised documentation (Glasson, 1984), and
- The need to improve the process by measuring its effectiveness (Wasserman *et al.*, 1983)

Most importantly, as far as this project is concerned, the development of a standard approach to requirements engineering makes it possible to consider generalised, automated computer-based tools, to assist in the process and increase its efficiency.

As already discussed, it is generally accepted that it is not possible to define with any consistency, a prescriptive process for requirements engineering as can be achieved with the software lifecycle for example. However it is possible to define the general characteristics which any method or process for requirements engineering should promote. Macauley (1996) summarises these characteristics of the requirements engineering process as follows:

- *Requirements Engineering Techniques*: The process must support a number of systematic techniques for problem analysis, modelling, documentation, quality and automated tools support.
- *Human Communication*: Amongst others, the process must adopt techniques for interviewing users, design of questionnaires, observing users, listening skills *and* supporting the users' view in model reviews. It should also clearly define the roles required in the process and the communication interfaces between them.
- *Knowledge Development*: The process should allow development of visions of design proposals and technological options, support knowledge of the current organisation and likely future changes.
- *Documentation Techniques*: Amongst others, the process should encourage the writing of unambiguous requirements, complete specifications, verifiable requirements, consistent requirements and support requirements traceability. In addition, it should encourage the development of models with a high level of abstraction, human readability, precision, completeness and support a mapping to a design technique.

In addition these areas must be supported by *Management Techniques* which focus on defining factors required to achieve the aims of the requirements engineering process, defining when the process is complete and managing the individuals involved.

2.2.1 Roles and Communication in Requirements Engineering

In general it is the human aspect of the requirements engineering task which introduces the most problems, and not surprisingly many requirements engineering techniques focus heavily on the roles of the individuals involved and the communication which takes place between them. As suggested by Macaulay (1996), four groups of techniques have emerged involving different levels of communication and user involvement:

- Techniques Promoting User Consultation
- Techniques Promoting User Participation
- Techniques Promoting Stakeholder Participation
- Techniques Promoting Stakeholder Co-operation

2.2.2 Techniques Promoting User Consultation

Perhaps the most traditional techniques for requirements engineering take the view that the requirements engineer is in control of the process, and that users are *consulted* to elicit detailed requirements. One of the most widely accepted methodologies for gathering and analysing requirements in this way is Structured Analysis. Structured Analysis has evolved over many years mainly due to the contributions by a number of researchers including Ross' SADT Model (Ross, 1977), DeMarco's Structured Analysis (DeMarco, 1978), McMenamin and Palmer's (1984) bottom-up approach to DFD's, Yourdon Structured Analysis (Yourdon, 1989) and SSADM (Downs *et al.*, 1988).

The approach advocates the use of Data Flow Diagrams (DFD's), in order to capture the flow of information between system components and thereby define the system interactions. Structured analysis defines a stepwise refinement process, which eventually forms a high-level logical design specification for the system, and as such it provides a useful method for partitioning the system requirements.

In recent years, Object Oriented Analysis (OOA) techniques have gained in popularity, supported by a number of methodologies and tools. Coad and Yourdon (1991) have

proposed a stepwise method for creating an object model of requirements and more recently requirements elicitation and analysis techniques such as OMT have been proposed by Rumbaugh *et al.* (1991).

Proponents of OOA techniques argue that the approach is simple to understand as it deals in real world entities or *objects* which in turn have attributes and behaviour. Objects are organised into *Classes*, which embody defined *methods* and *data elements* accessible to other objects by *contractual* interfaces. It is argued that OOA and OOD (object-oriented design) provide a framework for component-oriented models, in which data hiding is primary thereby allowing objects to be re-used. This is because, as the system evolves its functions tend to change but its objects in general do not (Davis, 1993).

OOA approaches are often referred to as *hard analysis, process or function-oriented* as they aim to improve the communication between the requirements engineer and the design team, rather than promoting the involvement of the user in the decision making process. Hence the user is consulted rather than being central to the requirements engineering process. As such, if the problem domain is well defined and the deliverables at each stage of the process are well understood, these can be very effective techniques. This view is supported by Flynn (1992) who proposes that because OOA is based on a real-world view of the system, a specification consisting of objects “*can be agreed upon more readily by all involved in the development process*”.

Indeed, SA and OOA are probably the most widely used analysis techniques, and have been implemented in a number of computer-based tools, for example Cadre's TEAMWORK tool (Cadre Technologies Inc, 1990). However, the depth to which the requirements engineer can understand and encapsulate the problem domain heavily influences the success of OO approaches.

A number of formal specification techniques for requirements capture such as VDM++ and Z++, which are based on OOA have been proposed by Lano (1995). In addition,

Semmens *et al.* (1992) and Semmens and Allen (1991, 1992) have reported results toward the integration of Yourdon structured analysis and the Z notation.

2.2.3 Techniques Promoting User Participation

Perhaps the earliest requirements engineering technique involving user participation was ETHICS (Effective Technical and Human Implementation of Computer-based Systems), a methodology originally proposed by Mumford and Weir (1979) and developed over many years since. It is a so-called *participatory approach* as defined by Avison and Wood-Harper (1991) in which the users of the system participate in its development, thereby increasing the chances of project success. The idea behind ETHICS is from a *socio-technical* viewpoint, which postulates that for the system to be effective the technology employed must fit closely with the social and organisational factors influencing the user. The methodology emphasises the need to ensure that improved quality of work and job satisfaction are a key concern in defining the required system.

Mumford argues this case by stating that the cause of many project failures is that the requirements process is driven purely by technical and economic concerns. As highlighted by Avison and Fitzgerald (1988), the socio-technical approach is characterised by Mumford as:

“one which recognises the interaction of technology and people and produces work systems which are both technically efficient and have social characteristics which lead to high job satisfaction”,

Another important contribution in this area is Soft Systems methodology (SSM) (Checkland, 1981), which evolved almost as a rebellion against the clinical, scientific or *hard* approach to systems analysis proposed by many in the early 1980's. Hard analysis is essentially *goal-oriented*, and aimed at defining an approach to achieve a given physical objective in the most efficient manner. In contrast, Soft Analysis assumes that there is more to the analysis than simply satisfying some arbitrary physical goal, and concentrates more on the *real-world* environment in which the system operates. The people and

objects with which the system will interact therefore heavily influence the approach. The system is also said to have a *mission* to improve the problem domain rather than to meet some arbitrary defined goal.

One of the main advantages of SSM is that it can be applied to a wide variety of problems, as the *rich pictures* technique is embodied is abstract enough to address almost any real-world situation. Thus it is more amenable to situations where the problem is ill defined or *fuzzy*, which is a recurring theme in many requirements engineering problems. Rich pictures are the main communication interface between the user and the requirements engineer. They will usually show people, controlling factors, conflicts and concerns, sources and sinks of information and relationships, which are usually captured in the terminology of the user. These pictures form a *Weltanschauung* or world-view from which the requirements engineer extracts problem themes. These themes are then used to define the overall requirements of the new system, called the *root definitions*. The root definitions, together with selection guidelines (termed CATWOE criteria) are then used to develop *conceptual models* of the system. These models are not intended as a direct representation of what the system currently does or what it ought to do, rather they are “*epistemological devices serving coherent discussion*” (Checkland, 1995). The models are then validated to ensure they represent a *viable human activity system*, and are then compared with what currently happens in the real world. From this comparison the requirements engineer then makes recommendations concerning change, and selects from these recommendations on the basis of feasibility and desirability. The final stage then suggests actions to address these recommendations.

The advantage of SSM is that it is a truly flexible approach and can be applied to many different problems. However, in terms of a *process* it suffers from several problems: Firstly it is difficult to define the skills required for the requirements engineer to use this approach and thereby train such a person. Secondly, it is unclear at which stage the requirements engineering process is completed, as there are no clear deliverables. Proponents argue that this is a good thing, as there are no pre-conceived *solutions* and the approach forces the requirements engineer to intimately understand the nature of the system. It has therefore been argued (Avison and Fitzgerald, 1988) that SSM is a good

front-end to the requirements engineering process, before moving into a hard methodology such as SSADM. Similarly, (Bustard and Dobbin, 1996) have proposed an approach to requirements engineering based on the integration of SSM and OOA, as a two-stage process combining the identification of business improvement with computer-oriented analysis.

Other so-called *participatory design* (PD) or *user-centred design* (UCD) techniques in which there is a strong user involvement in Requirements Analysis and Systems Design have been proposed by Floyd (Floyd *et al.* 1989), Bhabuta (1989) and Greenbaum and Kyng (1991).

2.2.4 Techniques Promoting Stakeholder Participation

Techniques in this category extend the notion of the customer beyond the concept of a simple end-user, to the concept of all those individuals who have a *stake* in the success of the delivered system. This involves those people directly responsible for the development of the system, those with a financial risk, those responsible for managing the change to the new system (e.g. installers, maintainers, trainers and support staff), together with those who will actually use the system. Such individuals are grouped under the generic term *stakeholders*.

As identified by Sommerville and Sawyer (1997), the aim of techniques in this category is to promote a *shared* understanding of the system being specified. If you do not consider everyone who is likely to be affected by the introduction of the system, important requirements are likely to be missed. In addition, consulting all stakeholders makes them feel part of the requirements elicitation process, and they are then more likely to be sympathetic to the introduction of the new system and hence volunteer more information about their requirements.

In the late 1980's and early 1990's a number of researchers (Hsia and Yaung 1988, Holbrook, 1990) introduced the concept of *scenarios* as an aid to promoting shared understanding of the system and validating stakeholder requirements. Scenarios are

intended to capture the required system environment as perceived by the stakeholders using elicitation, formalisation and prototyping techniques.

A *scenario* is a collection of partially ordered events, which define transitions from one system state to another. A scenario is initiated or invoked by an entity called an Agent, and each scenario defines an invariant condition, which must hold true throughout the scenario. By means of simulation and user interface techniques, scenarios allow a rapid prototype of the required system to be built, whose operation can then be validated by demonstrating the prototype to the users.

Several tools and methods have been constructed based on this approach including a *Screen-based Scenario Generator* (Hsai and Yaung, 1988), and a methodology termed *Scenario Based Requirements Elicitation (SBRE)* proposed by Holbrook (1990).

2.2.5 Techniques Promoting Stakeholder Co-operation

If it is agreed that the main role of requirements engineering is to maximise customer satisfaction with the delivered product, then the requirements engineering process can be considered part of an on-going *quality function*. This is the approach advocated by Zultner (1993), Brown (1991), Hauser and Clausing (1988) and others, where the emphasis is placed on the *voice of the customer or stakeholder*.

In these cases, requirements engineering techniques focus on providing a framework for discussion and decision making, with the emphasis on mapping customer requirements to product characteristics. Perhaps the most important technique in this category is Quality Function Deployment (QFD). QFD was developed around twenty five years ago, and originated at the Mitsubishi Kobe shipyard. Since then, other companies such as Toyota and AT&T (Brown, 1991) have developed it in numerous ways and have applied it to many markets including software development, consumer electronics, construction equipment and clothing manufacturing.

As defined by Brown (1991), QFD focuses on four strategic concepts:

- *Preservation of the voice of the customer.* QFD ensures that the customer needs are not translated or distorted in the development process.
- *Cross-Functional Realisation Teams.* QFD ensures that all areas of the business are included in the process and are given opportunity to air their views.
- *Concurrent Engineering.* QFD allows for those activities which would traditionally begin later in the development cycle to begin planning earlier, thus shortening the time-to-market.
- *Graphical Presentation of Information.* QFD focuses on a specific graphical tool termed the *house of quality (HOQ) matrix* (named for its apparent shape), which is a representation of the product which defines links to explicit customer needs and product realisation decisions.

Whilst the QFD method itself will not be described here, it is interesting to note some points which are explicit in QFD and are not highlighted in other requirements engineering processes.

Firstly, QFD focuses very much on time-to-market by recognising that the traditional approach to product development is highly sequential, involving a learning curve at each boundary. For example, the designers will need to learn about the specification of requirements when it is passed on from the requirements engineer, before work can begin. QFD highlights the need for all parties involved in the development process to be involved at the very start of the requirements engineering process.

Secondly, and perhaps paradoxically, the QFD approach advocates that not all requirements are equally important. It is argued that in traditional approaches, all requirements are treated equally and resources are stretched to try to address them all. In this way the same amount of effort may go into developing a less significant requirement, than an important one, and in this way the quality of the overall product is *diluted*. Instead emphasis is placed on those requirements which will maximise *customer satisfaction*.

These ideas are derived from Kano's model (Kano *et al.*, 1984) shown in Figure 2-3, which suggest that there are three types of requirements to attend:

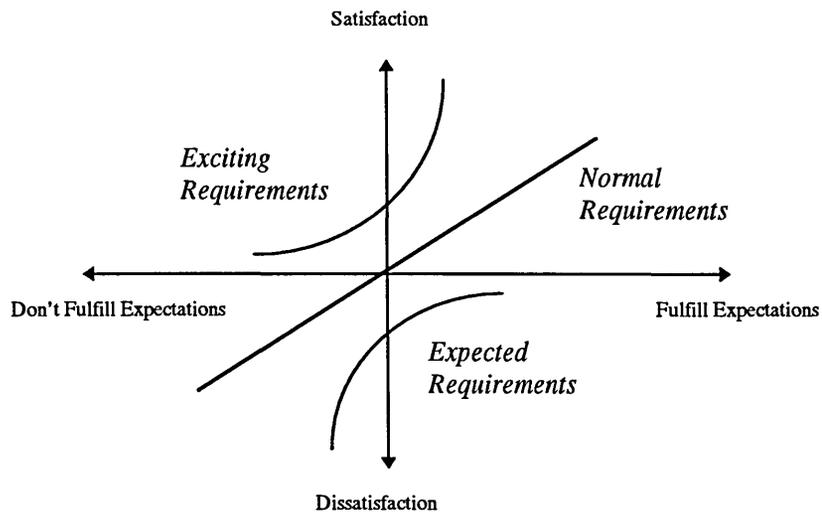


Figure 2-3: Kano's Model of Customer Requirements

- **Normal Requirements:** The difficulty with some requirements engineering processes is that they only ask the customer what they want. The easiest of these requirements to uncover are normal requirements. These requirements satisfy (or dissatisfy) in direct proportion to their presence (or absence) in the delivered system. An example of this type of requirement is the speed at which the system produces results; the faster (or slower) it is, the better the customer likes (or dislikes) the system.
- **Expected Requirements:** In order to avoid disappointing customers the system must deliver on some basic expectations which in many cases are not explicitly stated. Many customers find it difficult to articulate these requirements. Whilst their presence meets expectations, they do not themselves generate satisfaction. However, their absence is very dissatisfying to the customers. An example might be that the system requires some form of on-line help. This is so obvious for most systems that customers may assume it without explicitly stating the requirement. *Eliciting and meeting the expected requirements is a pre-requisite for successful requirements engineering.*
- **Exciting Requirements:** Perhaps the most difficult requirements to uncover are those that are beyond the customers expectations, yet are highly satisfying when delivered. The absence of these requirements does not generate dissatisfaction because they are not expected, however their presence generates deep satisfaction in the customer, and *excites* them about the system. It is suggested by Zultner (1991) that *“a truly successful system delivers at least a few wows!”*.

QFD has become popular in industrial applications as a requirements engineering approach, and several successes are documented (Brown, 1991). Advocates of QFD claim that understanding of the customer needs is improved and importantly, more of the customer's critical needs are met. Requirements initially missed are quickly uncovered, and enhanced communication and parallelism improve the speed of development of the product. Perhaps, most importantly, users of QFD claim that credibility with customers is enhanced by use of the process, thereby increasing the likelihood of future development contracts.

2.2.6 The Specification of Requirements

Whichever of the approaches described above is adopted, the objective of the requirements engineering process is the production of a document termed the *Specification of Requirements*. In general, it is not defined how this document is presented, however according to the IEEE (1993) a 'good' software requirements specification should contain requirements which are:

- *Correct* (the customer or user determines that the specification reflects their actual needs)
- *Unambiguous* (each requirements has one interpretation)
- *Complete* (all significant requirements, responses and terms are included)
- *Verifiable* (there exists some finite, cost-effective process which can check that each requirement is satisfied in the final product).
- *Consistent* (no subset of the requirements defined are in conflict)
- *Modifiable* (structure and style are such that changes can be made easily, completely and consistently).
- *Ranked in importance or stability*.
- *Traceable* (the origin of the requirement is clear, and can be identified in any subsequent development or documentation).

These objectives have some far-reaching implications for the production of the specification. For example, the term '*unambiguous*' implies that the document is presented in some format other than natural language. By definition, natural language is

open to interpretation by the reader since it has no formal semantics. The term '*correct*' implies that there is some way in which the requirements can be *tested* for understanding before the document is considered complete. The term '*modifiable*' again places a constraint on the format of the document to ensure that it can be maintained with minimal effort throughout its lifetime. There is also the implication that some form of *consistency checking* is built into the production of the specification. Hence it can be seen that the production of such a specification document is far from simple for any non-trivial problem.

The question is therefore, what mechanisms should be employed in the preparation of the specification of requirements to achieve these objectives?

Many researchers, too numerous to mention, have suggested ways of achieving these objectives, however it is generally accepted that the process leading to the specification of requirements can be divided into two parts (Davis, 1981). Firstly, the process of *eliciting* (Goguen and Linde, 1993) and *capturing* an unambiguous statement of requirements from the user, and secondly the process of *validating* that the set of requirements elicited represents a complete, consistent and correct representation of what the user requires.

One approach suggested in addressing this problem is to capture the requirements informally, and then transform the requirements definition into some formal notation based on discrete mathematics. Due to its mathematical and logical nature, such an approach emphasises precision and lack of ambiguity in the specification produced. The application of such techniques is known by the term *formal methods*.

2.3 Approaches Based on Formal Methods

Formal methods are perhaps becoming one of the most important fields of research in computer science. The last two decades have certainly seen an increase in the use of formal methods within the construction of software systems, but the rate and scope of their use in industrial application is still unclear. However, Vienneau (1997) asserts that

formal methods are becoming increasingly popular and are dramatically altering the way in which software is developed. As a note of caution, he himself admits that most organisations still have little experience of their use, and those which have are likely to have an SEI process maturity framework rating of three or above.

Leveson (1990), provides a definition of Formal methods as:

“a broad view of formal methods includes all application of (primarily) discrete mathematics to software engineering problems. This application usually involves modelling and analysis where the models and analysis procedures are defined by an underlying, mathematically precise foundation.”

The term *Formal Methods* essentially comprises two components: *Formal Specification* and *Verified Design* (Jones, 1990). The methodology underlying these terms is to first precisely specify the behaviour of the system in a formal notation, and then *prove* that the subsequent implementation is a true reflection of the original specification.

The early uses of formal methods concentrated on *proving* that an implemented software component met its specification, be it formal or informal. In general this is a non-trivial problem and is the subject of much research work in its own right, with early contributions from Hoare (1969) and Morgan *et al.* (1988). However, more recently attention has turned to the use of formal methods for the development of *formal specifications*, rather than formal design verification (Hall 1990, Place *et al.*, 1990).

Inherent in the use of formal methods for the purposes of *Formal Specification* is the primacy of *declarative* over *imperative* forms of thinking, in that the mathematics deals with notions of *what* the system should do, rather than specifying a prescriptive approach of *how* it should do it. This view does not suggest that formal specifications are any less precise. On the contrary, the formal specification can be viewed as a set of formulae in the chosen formal language, which describe a rigorous model of the desired behaviour of the system. Most importantly, the formal specification does not commit to details of how the model should be implemented. It is this *abstraction* from implementation detail whilst preserving essential properties of the system, which is the power behind the approach.

Several well-documented formal specification languages and notations have been developed over the years including:

- The Z Notation (Spivey, 1992)
- Communicating Sequential Processes (CSP), (Hoare, 1985)
- The Vienna Development Method (Jones, 1990)
- LARCH (Gutttag and Horning, 1993)

These specification languages can be classified (Wing, 1990) by their semantic domains (set of symbols and grammatical rules) as:

- Abstract Data Type Specification Languages (ADT)
- Process Specification Languages
- Programming Languages.

An ADT defines the formal properties of a data type without defining implementation features (Vienneau, 1991), examples being Z, VDM and LARCH. Process Specification Languages (PSL) specify state machines, event sequences, streams and partial orders, CSP being the best known PSL.

It is perhaps interesting to note at this stage that not all member of the computer science community share this faith in the application of formal methods to software development. A case in point is the view advocated by Abelson and Sussman (1985),

" The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the structure of knowledge from an imperative point of view, as opposed to the more declarative view taken by classical mathematical subjects. Mathematics provides a framework for dealing precisely with notions of "what is". Computation provides a framework for dealing precisely with notions of "how to".

However, it cannot be ignored that the tide of opinion is in favour of declarative methods of computing, as indicated by the plethora of formal methods notations and tools which have been developed over the years (Eisenbach 1987, Bowen and Hinchey, 1995).

2.3.1 Formal Specification Techniques

As highlighted above, Formal specification languages and notations have been researched over a number of years, and have grown out of the fields of program verification and program semantics. However, two important distinctions can be drawn in the way that they are applied (Avizienis and Wu, 1990).

Firstly, *Operational techniques* have been described as constructive or *model-oriented* (Wing, 1990). A formal specification based on operational techniques describes the desired system directly by providing a *model* of the system. Typically, this model uses abstract mathematical structures such as sets, relations and functions. In contrast, *definitional techniques* are described as *property-oriented* or *declarative* (Place 1990), and embody *algebraic* and *axiomatic* techniques in which the properties of the desired system are restricted to equations in certain algebras.

An early example of an operational, model-based approach is the Z notation, which was initially introduced by J.R Abrial in his paper “*Data Semantics*” in 1974. Since then the Z notation has been considerably expanded and enhanced by several researchers since the early 1980’s, particular by the Programming Research Group at Oxford University. The Vienna Development Method or VDM is another example of a model-based approach, and was also developed in the early 1980’s by IBM and later by Cliff Jones at the University of Manchester, This work resulted in the development of the Mural Toolset for the application of VDM (Ritchie, 1993).

Both Z and VDM have been the subject of much subsequent research including approaches to modularising Z specifications (Sampaio and Meria, 1990), and *object-based* extensions including HOOD (Iachini and Giovanni, 1990) and the B Abstract Machine Notation (Haughton and Lano, 1995). Since Z is a model-based approach it has

also been suggested that this naturally lends itself to object-orientation. Several *object-oriented* extensions to Z have been proposed including Object-Z (Carrington *et al.*, 1990), Z++ (Lano, 1991) originating from the ESPRIT REDO project, ZEST (Cusack, 1991), MooZ (Meira and Cavalcanti, 1991) and OOZE (Alencar and Goguen, 1991). VDM has also been the subject of object-oriented extensions culminating in the production of the VDM++ language (Durr *et al.*, 1994). This includes concepts from the original VDM (Jones, 1990), SmallTalk (Goldberg and Robson, 1983), the DRAGOON extensions to Ada (Atkinson *et al.*, 1991) and real-time extensions from Hayes (Hayes and Mahoney, 1992). In addition SmallVDM has also been developed (Lano and Haughton, 1993).

Whilst object-orientation is more in keeping with current approaches to system development, as Lano (1995) points out, several of these extensions suffer from a lack of formal semantics. This makes it difficult to reason about specifications written using these languages (however formal semantics and reasoning systems have been provided for Object-Z, Z++ and MooZ).

Alternative *definitional* approaches to formalisation include the use of Algebraic Notations such as OBJ (Goguen and Winkler, 1988), PLUSS and FOOPS (Goguen and Wolfrum, 1990). These use equational logic to represent implicit and abstract requirements within the system. Whilst they are easier to convert to executable forms (because of their limited language semantics), they tend to lack expressive capacity and thereby produce very algorithmic specifications akin to functional programming styles.

The key problem addressed by the use of formal specification languages in the requirements engineering process, is that of ensuring that requirements are unambiguous, complete and consistent. The main argument is that requirements captured in natural language or diagrammatic form are open to ambiguity and cannot express the semantic details of the required system with the same precision as a formally defined notation. These formally defined notations utilise mathematical concepts in order to define precisely the properties and constraints of the system to be designed. The implication is that if system requirements are captured in a natural language, these will contain inherent

ambiguity and therefore are a possible source of error in the later implementation phases. The aim of formal specification is to eliminate this possibility.

To assess the success of this goal, several researchers have assessed the applications of formal specification techniques in commercial development projects. Documented studies include:

- Formal specification and re-engineering of part of the CICS information system at IBM using the Z Notation (Collins *et al.*, 1988).
- Development of the floating point unit for the INMOS T800 transputer (May, 1990)
- Formal Specification and Test case generation for communication systems at British Telecom using the ZEST Notation (Cusack and Wezeman, 1993).

As reported by Lano (1995), benefits claimed include a 9% reduction in costs for the CICS system and a one-month reduction in time-to-market for the INMOS Transputer. Hence, there is some evidence that the application of formal specification techniques can lead to lower development costs since re-development and re-work is minimised, even though analysis and specification costs may themselves increase.

2.3.2 The Z Notation

Perhaps the earliest and most widely known formal specification notation is the Z notation, originally introduced by Jean-Raymond Abrial in 1974. Since then the notation has been developed by the Programming Research Group at Oxford University, and has now evolved to the point where it contains all the essential features necessary to address a large variety of specification problems.

The Z notation is a model-based, mathematical approach based on set theory and first-order predicate calculus, and includes a well-defined type system. Whilst the Z notation is well documented (King *et al.* 1988, Spivey 1989, Diller 1990, Potter *et al.* 1991) the definitive guide to the notation is that produced by Spivey (1992). More recently the *Z Base Standard* (Brien and Nicholls, 1992) has been produced in an effort to increase the

level of communication and portability of Z specifications, and provide a basis for CAE tools development.

A specification written in Z is :

" *a mixture of formal, mathematical statements and informal explanatory text. Both are important: the formal part gives a precise description of the system being specified, whilst the informal text makes the document more readable and comprehensible, linking the mathematics to the real world*" (Potter *et al.*, 1991).

The technique underlying Z is the use of *declarative modelling* without concern for efficiency or ease of implementation. The specifier describes the system in Z using *operational and representational abstraction*. That is, abstract mathematical structures such as sets, relations, functions, sequences and bags, for example, are used to characterise the desired behaviour of the system within logical predicates. These predicates and associated declarations are organised into *schemas*, which are the basic building blocks of a Z specification. A schema represents a logical specification unit, making it possible to divide the specification into manageable pieces. The contents of a schema should be self-explanatory without need to refer to other parts of the specification. Z also includes a powerful *schema calculus*, making it possible to construct new schemas by reference to others already defined. In addition, to aid understanding, a Z specification will also include sections of narrative text describing the purpose of the various components of the model it contains.

An extract from a simple Z specification defining a banking operation is shown in Figure 2-4. This extract demonstrates the use of mathematical objects to *model* elements of the banking operation in a formal way. For example, customer *ACCOUNTS* are modelled as a *partial function* which relates an account number from the given set *ACCNOS*, to the amount of money in that particular account (modelled as an element of the set \mathbb{N}). The fact that a partial function has been chosen for this purpose imposes additional mathematical formalism, for example each account number is related to *at most* one amount of money. To define *invariant conditions* and *operations* using this model, *variables* of the appropriate type are declared, and properties defined over them. It is

convenient to group related definitions into a single specification object termed a *Schema*. For example, in the extract shown in Figure 2-4, the *Bank* schema defines the invariant conditions that are to remain true in this *particular* banking operation.

```

[ACCNOS]
MONEY ==  $\mathbb{N}$ 
/* define a representation of a Bank Account as a function
   mapping an account number to an amount of money*/
ACCOUNTS == ACCNOS  $\rightarrow$  MONEY

/* Model a queue of transactions as a sequence of Account
   numbers */
QUEUE == seq ACCNOS

REPORT ::= withdrawalRefused

/* The main Bank schema defines an account and a queue */
┌── Bank ────────────────────────────────────────────────────────────┐
│ a : ACCOUNTS                                                       │
│ q : QUEUE                                                           │
└───────────────────────────────────────────────────────────────────┘

/* you can't be in the queue if you don't have an account */
ran q  $\subseteq$  dom a
/* your account must have some money in it */
 $\forall n : \text{ACCNOS} \mid n \in \text{dom } a \bullet a(n) > 0$ 
/* You can't be in the queue more than once */
 $\forall ij : \mathbb{N} \mid i \in \text{dom } q \wedge j \in \text{dom } q \bullet i \neq j \Rightarrow q(i) \neq q(j)$ 
/* The queue is at most 10 accounts deep */
#q  $\leq$  10

```

Figure 2-4: Example Z Specification

Z has been shown to be useful in specifying a wide variety of problem domains (Hayes, 1993), and has also applied to a number of industrial applications, perhaps the best documented of these being the attempt to re-specify IBM Hursley's highly successful Customer Information and Control System (CICS) described previously (Collins *et al.*, 1988).

2.3.3 Formal Specification and Bridging the Communication Gap

It is widely accepted that the use of formal specification languages and notations can help in eliminating ambiguity from the requirements engineering process and therefore uncovers errors at a much earlier stage in the project definition. Inevitably, the requirements engineering process is extended due to the inherent complexity associated with the mathematical notations. However, the gain in terms of the reduction in the amount of re-work and maintenance far outweighs the extra work at the front-end of the development cycle.

Yet, in previous discussion we have made much of the need for requirements engineering techniques to promote stakeholder participation and communication. If we analyse the *contribution* of requirements engineering techniques based on formal methods, we find that they are strong in the areas of process definition (having well defined semantics and scope), but are weak in the area of human communication. In general this is due to the highly mathematical notations involved, and the consequential specialist knowledge required in interpreting them.

In turn this has led some commentators to propose that the mathematical content of formal specifications prohibits effective communication during the requirements engineering task. As highlighted by Saiedian (1997), this negative perception of the role of mathematics in requirements engineering is unfortunate. In other disciplines, engineers naturally turn to mathematics for assistance when large, complex problems arise. However, many software engineers take the view that formal methods are for academic interest only, and that real problems are too complex to be handled by mathematical

tools. Cherniavsky (1990) supports this view in a report released through the U.S. House of Representatives Committee on Science, criticising inadequate education for software engineers.

However, the lack of adequate education in formal methods for software engineers is only one problem. More importantly, the formal specification forms a *contract* between the developers and the Stakeholders of the required system. The Stakeholders are even less likely to possess the necessary mathematical skills to *validate* that the specification is indeed a true statement of the system requirements.

Hence, this problem presents us with a dichotomy: How is it possible to make use of the obvious benefits of formal techniques in improving the quality of the system specification, whilst at the same time bridging the communication problems which accompany the use of a mathematical notation?

It is proposed that one solution to this problem is the use of *Rapid Prototyping and Animation of Formal specifications* for the purposes of *validation by execution* as discussed in Chapter 1. However, to achieve this by the most efficient means requires either that the specification itself supports direct execution within a computer system, or that it can be transformed into a representation capable of being executed in some animation environment. This in turn has lead researchers to investigate the possibilities for *executable specifications*.

2.3.4 Related Research into Executable Specifications

The use of formal methods allows us to *verify* that a particular implementation satisfies some formal specification. However, the use of a formal methods approach does not in itself *validate* that the system specification has met the perceived need. That is, the use of formal specification does not provide a mechanism to prove that the specification captures the user's perception of the required system.

To achieve this goal it is suggested that rapid prototyping of the formal specification is required to allow the user to explore the behaviour and properties of the specification directly. The most efficient means of achieving this is by *executing* the specification directly although, in the case of a formal specification, this in itself may present several problems as discussed later. These problems arise since, depending upon the particular formal language chosen a formal specification may contain *non-computable clauses* that inhibit direct execution (Hayes and Jones, 1989).

In general, executable formal specifications bridge the gap between the traditional approach to software prototyping and the application of formal methods. That is they supply a direct relationship between the rapid prototype of the system and the formal specification documentation.

As Fuchs (1992) describes, an executable formal specification can be regarded as an abstract program which allows abstract requirements and designs to be formulated, explored and validated at an early stage of software development. Using this approach the system interaction with its environment can be demonstrated and observed, preserving the link to the formal specification documentation. Indeed, in some cases, the executable specification may form the only relevant document for all development phases, such as the use of executable specifications within transformational approaches (Berzins *et al.*, 1993).

One of the earliest approaches advocating the use of formal specifications for the construction of prototypes was that of Goguen and Meseguer (1982), who suggested the use of the algebraic specification language OBJ, combined with a system of equational interpretation. Since then, there have been several approaches proposed to execute formal notations for the purposes of rapid prototyping. These approaches divide into three broad categories:

- Approaches Using Declarative Programming Paradigms
- Proprietary Executable Specification languages and Code Generation Tools
- Environments to Support automatic prototyping of specifications

Firstly, the use of declarative programming paradigms (including functional and logic programming), has been widely researched (Turner 1985, Henderson 1986, Kowalski 1985). Proponents argue that declarative programming techniques combine the clarity required for a formal specification, with the ability to validate by execution. As such they are ideal for rapidly prototyping a design *as it is developed*. Indeed, the prototype is also effectively the final implementation since the languages used coincide.

A logical extension to this idea, investigated by several researchers is to provide a translation from a model-based formal notation such as Z or VDM into a functional or logical based programming language. In particular, Hekmatpour (1988) suggested extensions to LISP to achieve this. In addition, Johnson and Sanders (1990) have described the transformation of Z into functional implementations, O'Neill (1992) has shown how the ML language can be used to prototype VDM specifications, and Sherrel and Carver (1993) have researched the translation of Z into Haskell. In addition, Knott and Krause (1992) have used program transformation systems to implement Z specifications. The PROLOG language has also been extensively researched as a prototyping language by the likes of Dick *et al.* (1990). Indeed, the starting point for this project concerns the use of the LISP language and the provision of suitable extensions to transform Z specifications into executable prototypes (Siddiqi *et al.* 1991, Morrey *et al.* 1992).

The second category of note embodies research into “proprietary” executable specification languages designed specifically to embody executable semantics. In general, these languages are either embedded in an existing programming language which in turn provides the execution mechanism, or are translatable directly to a programming language. Notable research in this area includes:

- the work of Henderson and Minkowitz (1985) in the development of the Me-too language embedded in LISP
- the ASSPEGIQUE environment (Bidiot and Choppy, 1985) for the development of large algebraic specifications
- the GIST specification language (Balzar, 1985), and

- the OBSCURE language (Lehmann and Loechx, 1987) which is intended as an executable specification language independent of the specification method used.

There are also a number of systems capable of translating abstract data type specifications into executable programs (Belkhouche and Urban 1986, Jalote 1987, Bergstra *et al.* 1989). In addition, a number of results have been reported in the development of C++ translators for Z++ (Lano, 1991) and VDM++ as part of the AFRODITE ESPRIT project (Lano, 1995). Perhaps the most effective code-generation tools based on formal methods currently in existence are those for the B Abstract Machine Notation within the B Toolkit (B Core UK Limited, 1994). In addition the RAISE toolkit provides C++ and ADA translators for the RSL notation (George and Prehn, 1992).

The final area of research is that characterised as the development of environments for automatic prototyping of specifications. The initial work for this project (Siddiqi *et al.*, 1991), was influenced by the work of Hekmatpour and Ince (1988), who developed the EPROL language (based on VDM), to facilitate the execution of VDM specifications using constructors implemented as extensions to LISP. Work allied to that presented herein includes the automatic prototyping of Z (Doma and Nicholl, 1991), and VDM-SL (Elmstrøm *et al.*, 1994). Others, such as Valentine (1995) have focused on producing a *computational subset* of Z, as in the Z-- language.

2.4 Summary

In this chapter, the background to requirements engineering, formal specification and executable specifications has been explored.

The case has been made for the use of formal specification techniques to ensure that the specification of requirements is concise, precise and unambiguous. However, it has been noted that the use of formal specification techniques introduce *communication problems* as the user is unlikely to be able to *validate* that the specification as presented is indeed a

correct statement of the requirements of the desired system. This is due to the specialist mathematical notation employed.

It has been proposed that rapid prototyping of the system specification may assist in this process by allowing the user to explore the behaviour and properties of the formal specification directly, thus validating the specification by execution. However, to maximise the efficiency of the approach demands that such rapid prototypes be generated directly from the original specification by some automated tool. This in turn has led to suggestions for *executable specifications* and *animation environments*, requiring that a tool be able to produce an executable representation directly from the system specification

However, in order to design an automated tool to achieve these objectives, it is first necessary to develop a requirements engineering *process* into which the tool will integrate. In addition is it necessary to understand the *relationships* between different tools that can address the requirements engineering task, and also the *value* that such tools might be in assisting the people directly involved. In other words is it necessary to understand the interplay between requirements engineering tools and the stakeholders involved in the requirements engineering process. These issues are explored further in Chapter 3.

3. Requirements Engineering Tools and Processes

It has been suggested (Saiedian, 1997) that one of the main factors limiting the use of formal methods is the lack of investment in automated tools to reduce the effort involved in applying them. In contrast, a key factor in the acceptance of high level languages for development purposes is the presence of a comprehensive set of tools to support their use. It follows that if formal methods are to achieve the same level of acceptance within industrial software development organisations, they too require a similar level of automated tool support.

However, before beginning to describe the foundation for the development of a formal methods toolset, it is necessary to consider the needs of the requirements engineering task itself. Too often in complex development programmes, toolsets and methodologies are chosen purely on the basis of the skill set of the *developers*, without much thought to the needs of the *development* itself. Traditionally, the choice of a development toolset is dictated by factors such as:

- Existing or legacy software development processes,
- The human skills available,
- The availability of existing tools,
- stipulation by the system sponsor or external accreditation bodies,
- timescale and budgetary considerations.

However, the selection of a toolset based purely on the availability of existing systems, or rigid development processes may have dire consequences for the success of the project. For example, an organisation may stipulate that Object Oriented techniques are to be used for all developments, even though a particular problem domain may not naturally embody semantics making this approach effective or even practical. Toolsets must therefore be chosen based on the problem domain itself, and consequently tools suppliers must understand how their tools support the *processes* involved in solving such problems.

As described previously, the *classical* approach to Requirement Engineering has developed with no real underlying processes. This is due to a number of factors discussed previously in section 2.2. The main reason for this is that requirements engineering is *situation-oriented*, the process involved being dictated by the nature of the problem domain being addressed. Due to this problem it is generally accepted that the requirements engineering task is normally associated with a three-phase *approach* based on:

- *elicitation*: The process of forming an understanding of the needs of the system.
- *capture*: The process of documenting requirements in some defined format.
- *validation*: The process of confirming that the captured requirements are a true statement of need.

Yet this classical approach to requirements engineering raises some important questions:

- Is it possible to define individual processes associated with these three phases?
- If so, which components of these processes can be assisted by automated tools?, and
- What features should be implemented in these tools to add value to the process?

This chapter considers the issues associated with establishing how automated tools can assist within the requirements engineering task. This is achieved by the development of a requirements engineering tools hierarchy, based on an analysis of the concerns of the stakeholders involved in the task. Having established this hierarchy, the activities within the requirements engineering task are explored in order to derive a set of *product requirements* for a tool supporting formal specification in Z. Finally, the chapter concludes by defining a requirements engineering *validation* process into which the TranZit tool will be integrated.

3.1 Developing a Requirements Engineering Tools Hierarchy

The first question to consider is what is the perceived need for a requirements engineering tool?

As Brackett (1990) describes, users select tools based on their perception of the single expected major source of complexity in the system to be designed. For example, whether the system has many co-operating functions, has a complex data hierarchy or has particular control characteristics such as real-time interactions. On the basis of this analysis, a user might then select Cadre's Teamwork tool (Cadre Technologies Inc, 1990) for SASD to address functional complexity, or iLogix Statemate Tool (Harel, 1987), for Statechart descriptions, depending upon the characteristics of the perceived complexity.

Such tools embody a well-defined process framework, and as such if used correctly they will add some value to the development process. Similarly, if it is to be useful, a requirements engineering tool must add some value to the requirements engineering task.

As Jarke and Pohl (1994) describe, in the early characterisation of the requirements engineering task several fundamental assumptions were made:

- that there exists a well-defined problem, that can be clearly scoped and described,
- that the system specification forms the basis of a contract between the user and the developers,
- that each problem is different from others,
- that users are typically computer illiterate, but are domain specialists, and
- that methods used in requirements engineering are generalisations of methods used for systems development.

Whilst it is still accepted that the specification is the basis of a contract between the user and developer, experience has shown that many of the other assumptions no longer hold true.

In the first instance, it is recognised that in many cases no well-defined problem exists at the beginning of the requirements engineering task. As has already been discussed, users are much better at criticising existing systems rather than articulating their needs from first principles, and this may present serious problems for the requirements engineer

during the *elicitation* phase. Although Goguen and Linde (1993) have done much work in establishing *techniques* for requirements elicitation, the process involved is largely dictated by the particular problem domain and the experience of the individual requirements engineer. This in turn makes it difficult to devise automated tools that can add value to the elicitation phase itself.

Secondly, it is no longer the case that users are computer illiterate. As such the expectations that users have at the outset of a software project are correspondingly higher than at any time in the past. As described by Lubars *et al.* (1993), this has caused the requirements engineering task to expand beyond traditional problems which are *customer or service focused*, towards a need for *market driven requirements engineering* in which the components of the output specification are offset against the needs of the marketplace. Thus the requirements engineering task is moving away from compartmentalising problems which can be addressed purely by traditional computer science techniques, into more *real-world* models which correspondingly describe more abstract entities such as organisation structures, communication mediums and working practices.

It is also the case that the rate of technological advances continues to increase year on year, and that in order for systems to survive they must be able to deal with the pressure for continuous change. This is not only a consequence of the increase in technology availability but also economic considerations, a prime example being the rapid increase in E-Commerce.

All these issues place further emphasis on the requirements *capture* process, which must be sufficiently abstract and flexible to assimilate complex system descriptions both at project inception and also as the system evolves.

Finally, the need to be able to *validate* that the captured specification of requirements is indeed a true representation of the desired system is paramount in ensuring that the right product is developed and that user requirements are met, thus avoiding costly re-work.

Based on these issues it is possible to identify general considerations for requirements engineering tools as follows:

- There is a need for requirements engineering tools to capture a *worldview* of problems, rather than a constrained or system-oriented view. In this context they must be able to model abstract entities, rather than simply the technological (functional or data) components of a required system.
- Any requirements engineering tool must deal with the need for continuous change
- Requirement engineering tools must be able to partition problems into manageable units. In particular with object-oriented techniques increasing in popularity, the need to be able to *re-use* specification components is likely to be a desirable property in the future.
- There is a need for requirements engineering tools able to demonstrate a working model of the required system for the specifier to be able to *validate* the specification with the sponsor. Ideally, the tool should aim to hide the underlying computer technology as much as possible. The sponsor may then agree or disagree with the model as required, or may mitigate certain requirements on the basis of cost or market need.

3.1.1 Identifying the Users of Requirements Engineering Tools

The next consideration is that of identifying:

- Who are the potential users of requirements engineering tools?
- How do users expect requirements engineering tools to add value to their contribution to the requirements engineering task?

It follows that the users of requirements engineering tools will potentially be any of the stakeholder roles associated with the project development. In this respect we are dealing with the following roles:

1. *The end users.* The set of people who will operate the system on a daily basis.

2. *The system maintainers.* The set of people who will support/develop the system in the future.
3. *The system developers.* The set of people concerned with developing the completed system.
4. *The development management.* The set of people concerned with delivering the completed system.
5. *The customer.* The sponsor of the system. This may either be an external customer or an internal customer such as the marketing department originating the system request.
6. *The dependants.* The set of people who do not directly use the system, but who's well being may depend on it.
7. *The establishment.* The set of people who may have existing or developing systems which are required to inter-work with the new system.

Each of these stakeholder roles will present a different perspective or *viewpoint* to the requirements engineering task.

The impact of viewpoints on the requirements engineering task is well documented by Sommerville and Sawyer (1997), who suggest various approaches (such as PREview) to ensure that all viewpoints are considered. However, taking a slightly different perspective and introduce practical experiences, we can expand these ideas to deduce whether a requirements engineering tool is likely to impact on the effectiveness of a particular stakeholders contribution to the overall task.

The major stakeholder viewpoints or concerns are summarised in Table 1. This table has been established by researching the behaviour of stakeholder teams addressing real-world product development issues in an industrial environment within a major development organisation. The stakeholder concerns are ranked by importance:

Stakeholder	Primarily concerned with (in priority order)
End User	Functionality (and Usability) Performance Reliability Expandability Interoperability
Maintainer	Reliability Debugging, maintenance and training aids Interoperability Expandability
Developer	Functionality Performance Portability Reuse
Management	Costs Resources Schedule
Customer/ Marketing	Functionality (and Usability) Costs Schedule Interoperability Performance
Dependants	Reliability performance
Establishment	Interoperability Reliability performance portability

Table 1: Stakeholder Concerns

It can be seen that the primary concerns intrinsically polarise the stakeholder team into three distinct groups.

- 1) The group consisting of the User, the Customer (especially in the case of market-led products) and the developer who are primarily concerned with *the system functionality (and the dependent goal of usability)*.
- 2) The group consisting of the maintainer, the dependants and the establishment who are primarily concerned with the *system reliability and maintainability*.
- 3) The management (and to some extent the customer) who are primarily concerned with *system costs and schedule*.

The behaviour characterising each of these groups has been observed to some degree in all stakeholder teams researched. Not surprisingly each stakeholder presents *a viewpoint* which is primarily concerned with the factors which most affect the likelihood of achieving the stakeholders functional objective. However, from the perspective of tools development, this research can also be used to identify concerns and problems within the requirements engineering task which can be addressed by tools.

3.1.2 Identifying Requirements Engineering Problems from the Primary Viewpoints

From the preceding discussion, using these three primary viewpoints it is possible to deduce a taxonomy of requirements engineering problems:

- Problems associated with the description of *system functionality and usability* ,
- Problems associated with ensuring *system reliability and maintainability*,
- Problems associated with controlling *system costs and project schedule*.

Each of these requirements engineering problem groups offers an opportunity for automated tools to assist the appropriate stakeholders in achieving their given objectives. However, what are the characteristics of such tools to address these problem groups, and what value can they add to the requirements engineering task?

3.1.3 Tools to Control the Relationship Between Requirements and System Costs

Firstly, consider a hypothetical tool that helps in evaluating how a set of requirements impact on the system costs and project schedule.

The premise here is that *all requirements have a cost of development*. The issue is whether that cost is justified given the contribution that achievement of the requirement would bring in terms of overall customer satisfaction. Such *market-led* requirements analysis, effectively rank requirements in order of desirability and cost/benefit analysis. However, having made this ranking it is necessary to have an objective measure of whether the ranking is justified, or even sensible in the context of other requirements in the system (for example, whether system performance can be increased at the expense of portability). Since these relationships can be quite complex, and in some cases rely on expert knowledge, a tool to store these relationships and evaluate the judgements made would be very useful in resolving conflicts.

In their paper *Quality-Requirements conflicts*, Boehm and In (1996) describe a tool to assist in this task, terming it the *Quality Attribute Risk and Conflict Consultant* (QARCC). QARCC is a knowledge-based tool, which captures the relationships between stakeholders and their primary concerns (or *quality attributes* as Boehm terms them), how these quality attributes are manifested in the required system, and the strategy introduced to describe the relations between them. The system uses a *negotiation model* as its core component, in which stakeholders identify their desired quality attributes. The system then uses a knowledge base to identify software architectures or strategies for achieving that goal. The system then uses another part of its knowledge base to identify any potential conflicts in the quality attributes if this strategy were to be employed, and then provide suggestions as to how to resolve them.

A similar tool for resolving ‘interference’ between viewpoints, based on the Semantic Index System (SIS) is described by Spanoudakis and Finkelstein (1997).

In general, such tools aim to add value to the requirements engineering task by improving the management of people and resources involved in the development.

3.1.4 Tools Which Capture Requirements for System Reliability and Maintainability

The question here is what are the characteristics of a requirements engineering tool which will show that the set of requirements developed will produce a reliable and maintainable system?

Requirements involving terminology such as reliability and maintainability come under the general term of *Non-Functional requirements* or NFR. In practical terms, it is not enough to simply fulfil the *functional requirements* of a system. The system subsequently developed might work as intended, but it may be difficult to use or may exhibit poor performance under particular conditions (User-centred NFR). In addition it may be difficult to maintain or modify (developer-centred NFR) (Ebert 1997). Typically NFR's include:

- performance factors
- reliability factors
- usability factors
- System limitations and degradation criteria
- maintainability
- extendibility
- operational correctness (a measure of the extent to which the software satisfies the specification of requirements; especially in safety-critical applications).

Whilst the fundamental distinctions between functional and non-functional requirements have been well documented (Davis 1993, Myopoulos *et al.* 1992), there is still no comprehensive quality measure for ensuring that non-functional requirements are captured and implemented in the final system. As highlighted by Ebert (1997), this is largely due to the fact that in general, implementation of NFR's cannot be qualitatively

measured during the design phase. Rather we must wait until the System test and acceptance phases to see how the system performs in the real target environment. Although several simulator tools have been developed to model such characteristics as performance prior to design (e.g. MODSIM), they can only provide a limited level of confidence, since they intrinsically rely on empirically defined information on which model assumptions are based. Similarly, prototyping tools are also likely to be of limited use, since they are explicitly based on an inefficient model of the system, which is unlikely to exhibit the performance characteristics of the delivered system.

Hence, the majority of work in this area of requirement engineering tools has focused on *traceability* of requirements. Traceability is concerned with showing that a particular requirement has been implemented by some component of the delivered system (Ramesh *et al.*, 1997), and also in proving implementation correctness. In addition, some work has been done to produce automated test case generators, which can produce test suites directly from specifications. These can then be used by developers to test the final implementation (Richardson *et al.*, 1992).

3.1.5 Tools Which Capture Requirements Associated with System Functionality

The question here is what are the characteristics of a requirements engineering tool which will show that the set of requirements captured is a true representation of the required system functionality? In this context we refer to functional rather than non-functional requirements.

This is probably the area in which tools support has traditionally focused within requirements engineering, largely because the system functionality is a quality appreciated to a greater or lesser degree by all the stakeholders. It is also the case, that the challenges offered in attempting to capture the System functionality are amongst the better-understood problems that can be addressed by computer-based tools.

Areas in which tools can assist in capturing the system functionality are:

- Assisting in the manipulation, control and management of specification documents, which embodying the system functionality. This includes document formatters, Pretty printers and layout tools such as L^AT_EX (Lamport, 1985) for the Z formal notation (Spivey, 1992). In addition, there has been much work done in producing tools assisting in the translation of informal specifications (such as those written in natural language) into languages with more precise semantics. These are well documented by (Vadera and Meziane, 1997).
- Structuring requirements into organised, logical groupings and providing a basis for traceability. An example of a commercial tool in this context is DOORS (QSS Inc, 1998),
- Analysing the internal consistency of captured specifications. Many of these tools are dependent on the particular specification language used. For example, the TranZit tool described herein provides a syntax and type checker for the Z notation to ensure that the captured specification is both syntactically correct and its objects are of the correct type.
- Assisting in the *validation* of captured specifications. Whilst parsers and type checkers can check the internal consistency of specifications for language usage, they cannot in themselves show that the contents of the specification describe the customers requirements. Such tools are said to address the area of *Specification Validation*.

3.1.6 Tools for Specification Validation

The development of automated tools to assist in the process of specification validation has drawn much research interest in recent years.

Specification validation is the final phase of the general requirements engineering task, which is concerned with determining whether the specification is correct in some sense (Barden *at al.*, 1994). Inevitably, this work has grown out of the definition of languages in which to capture specifications, and as previously discussed many of these embody formal semantics which can be analysed by computer to a greater or lesser degree. To

this extent, automated tools research into validating specifications has focused on *Theorem provers, Model Checkers and Animation Engines*.

Theorem provers help to reveal what is implicit in specifications rather than explicit (Ciancarini *et al.*, 1997). For example, a theorem prover for a formal specification language such as Z may automate the computation of *pre* and *post* conditions over operations, and verify these against declared system *invariants*. The aim is to increase the quality of the specification and thus increase the confidence in its correctness. Whilst a formal proof for a specification is a worthwhile goal, even with tools support, in many cases the effort involved in generating the proof may well exceed the effort of actually capturing the specification itself. Thus, as Jackson (1994) points out, a value judgement must be made in assessing the need for a formal proof against the cost of actually performing one (for example, if the specification relates to a safety-critical system). Automated Theorem prover Tools such as Z/Eves (Saaltnik, 1989), the Larch Tools (Guttag and Horning, 1993) and the integration of Z with HOL (Higher Order Logic) (Bowen and Gordon, 1994, 1995), have been well documented. However, they are often criticised for the amount of user involvement and level of mathematical understanding required to make effective use of them.

In contrast, model checkers do not require user involvement, but inevitably constrain themselves to a particular requirements capture approach. A particular tool of interest is *NitPick* (Damon and Jackson, 1996), which is a model checker based on the Z notation capable of performing exhaustive analysis of finite state machines.

Finally, animators and execution engines attempt to transform (generally) non-procedural specification representations into some executable form. This approach usually involves the use of rapid prototyping to build an inefficient, yet functionally complete executable representation of the system embodied by the system specification. Animators are seen as a practical alternative to formal system provers, where the cost of a formal proof cannot be justified. Using the executable prototype, the stakeholders can explore properties of the system embodied by the specification, to verify that it exhibits the required behaviour.

Again, the premise is that stakeholders are much better at criticising existing systems than articulating their needs from first principles.

Several purists (Hayes and Jones, 1989) argue that since the conversion of an abstract (non-executable) representation to an executable one is generally *non-computable*, there is no merit in attempting to pursue this line of tools research. In contrast it has been argued in this work (Siddiqi *et al.* 1998, Morrey *et al.* 1994, 1998) and by several other researchers (Dick *et al.*, 1990), that it is possible to develop animation tools which have demonstrable, practical benefits in the area of specification verification.

3.1.7 A Requirements Engineering Tools Hierarchy

From the preceding discussion, it is now possible to define a tools hierarchy for Requirements Engineering as shown in Figure 3-1.

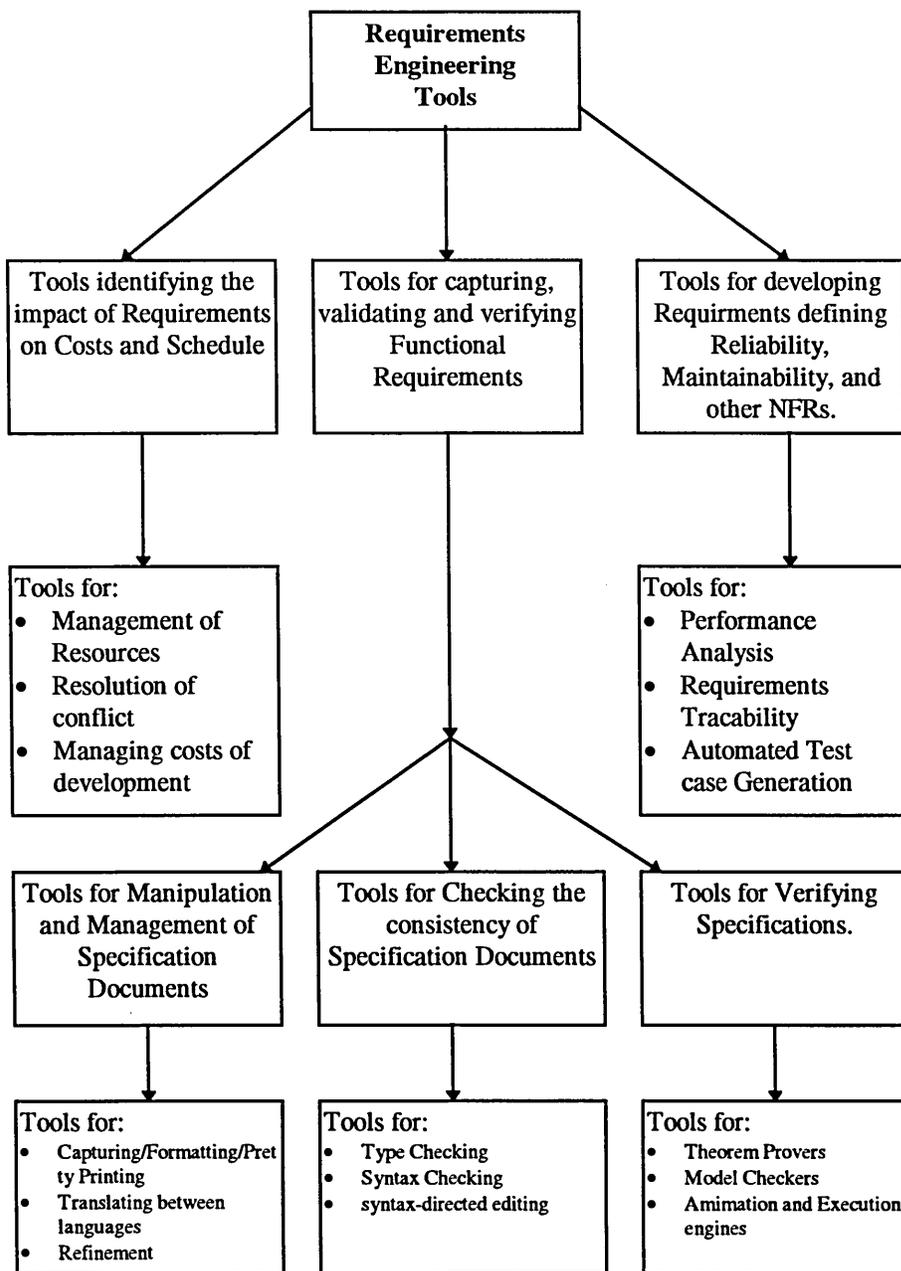


Figure 3-1: Requirements Engineering Tools Hierarchy

Whilst most tools choose one of the three main branches as a starting point, in general a tool may implement different features to a greater or lesser degree as the branch descends. For example, most functional requirements definition tools will commonly implement some form of capturing/formatting interface together with consistency checking functions such as a syntax analyser. On the other hand, there are well-

documented tools such as ZTC (Jia, 1994), (a pure type checker for Z), which defer the job of capturing and formatting the specification to other tools such as L^AT_EX (Lamport, 1985).

3.1.8 Requirements Engineering Tools Limitations

It is important to recognise that a tool cannot solve all the problems associated with the requirements engineering task. A tool exists to *assist* the stakeholder team in arriving at their goal. For example, a tool cannot (as yet) really assist in the human communication process of requirements acquisition and elicitation. Although there are many well-documented approaches to addressing the issue of requirements elicitation (Goguen and Linde, 1993), it is still largely down to the experience of the requirements engineer to adopt appropriate techniques on the basis of individual discussions. Perhaps in the future it may be possible to address this issue with knowledge-based AI tools, although this area requires more detailed research.

3.2 Developing Requirements for a Requirement Engineering Tool

So far this chapter has identified the likely users of requirements engineering tools and the particular viewpoints that these users present. This information has been used to define a taxonomy of requirements engineering tool types, which address the three primary viewpoints. These are:

- Tools for capturing system functionality and usability in a high quality and verifiable specification document.
- Tools for capturing non-functional requirements addressing system reliability, performance and maintainability
- Tools which analyse competing or conflicting requirements and derive associated system costs and scheduling information for project management

Having identified the *characteristics* of requirements engineering tools, it is now necessary to consider specific requirements for a tool to assist in the requirements engineering task. To achieve this it is first necessary to appreciate the *component*

processes of the requirements engineering task, with a view to establishing the specific requirements that these processes place on tools to support them.

Pohl (1993) describes the component processes of the requirements engineering task, using the analogy of three-dimensional space as shown in Figure 3-2. According to Pohl, the process of refining requirements and the subsequent degree of completeness of the specification forms *the specification dimension*. Similarly, the process of decision making and the subsequent degree to which the stakeholder team agree that the specification is a true representation of the system requirements is termed the *agreement dimension*. Finally, the process of capturing requirements and the subsequent degree to which the requirements are technically described using formal semantics is termed the *representation dimension*.

Using this model, any particular requirements engineering process can be described by tracing its path in three-dimensional space using the associated co-ordinate system.

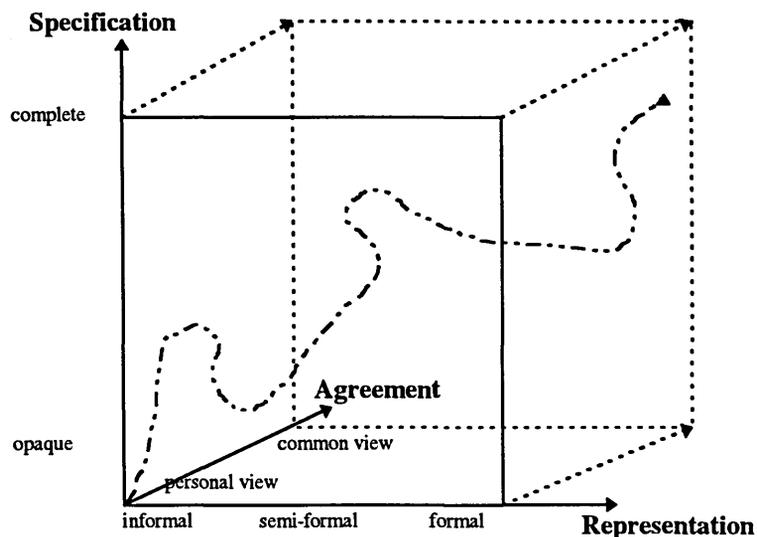


Figure 3-2: Pohl's Three Dimensions of Requirements Engineering

As Jarke and Pohl (1994) describe, the overall requirements engineering process normally begins near the origin of the framework. This point represents the state of the

process at project inception, characterised by differing personal views of the system held by individual team members, opaque system understanding and an informal representation. The goal of the process is to reach common agreement on a well-understood and complete specification, captured using formal semantics. In the course of achieving this goal, the track taken through the model describes the way in which particular issues relating to the problem domain are resolved by the requirement engineering process adopted. Clearly, the shortest route to achieving this goal is a straight line, implying that an effective requirements engineering process should maintain a balanced view all three component processes.

Therefore, to maximise the *value* added to the process by a requirements engineering tool it is clear that any tool should aim to support the *agreement* process, the *specification* process and the *representation* process in equal capacities.

3.2.1 Developing Requirements for a Tool to Support the Representation Process

The first question is what are the requirements for a tool to assist in maximising the efficiency of the *representation* process? In essence, the particular language chosen to capture the specification largely influences these requirements. The goal of the representation process is the production of a specification that is:

- complete
- implementation independent,
- unambiguous, precise and internally consistent,
- verifiable
- modifiable
- understandable by all stakeholders
- organised with built-in traceability.

It is interesting to note that in addition to these generally accepted properties of a quality specification, (Davis et al., 1993), add further properties as follows:

- Electronically Stored
- Executable/Interpretable/Prototypeable

- Re-usable

The most obvious choice for a representation language is natural language itself, since this is the mode of communication we use in everyday life. However, natural language specifications suffer from some serious drawbacks. Traditional specifications represented in natural language tend to run anywhere from a single page to many thousands of pages. However, as Davis (1988) points out, the size of the specification rarely has any relationship to the complexity of the problem. Rather, the inherent ambiguity of natural language statements means that in order to introduce completeness, the specification must be written in more verbose terms. This is not to say that natural language is not a powerful abstract communication medium; it is simply the case that it embodies ambiguous semantics. Similarly natural language does not embody any descriptive formalism. A particular reader may deduce any number of subjective implications from a simple natural language statement, which may or may not be true. Whilst this problem is well recognised, in their attempts to resolve the problem, natural language specifiers may well produce over large and complex documents which are difficult to assimilate and verify. Hence, it is suggested that natural language is unsuitable as the basis of a representation process for requirements engineering because,

- it is inherently ambiguous,
- it contains no formalism leaving the reader free to make subjective judgements about the meaning of the specification,
- it is difficult to check the completeness of a natural language specification,
- natural language specifications are inherently complex, the complexity increasing with attempts to resolve the problems highlighted previously.

All told, whilst the use of natural language meets the important criteria for specification readability and intelligibility, it should be clear that there is a need to introduce formalism into the representation process to address the important issues of completeness, ambiguity and verification.

3.2.2 Introducing Formalism Within the Representation Process

By *formal* we mean the introduction of a language into the representation process which has unambiguous descriptive semantics, and about which we can reason to deduce properties of the specification. In this way, the major goal of defining the system requirements in such a way that there is only one interpretation can be achieved.

Secondly, as discussed later, one of the major problems identified is the need to maintain the *interfaces* between individual specification objects in a coherent and cohesive manner. This is associated with the need to group related requirements together, reducing the burden on the reader in retaining a number of different concepts in foreground memory simultaneously. This is difficult to achieve with natural language, since there is no concept of a specification object, nor any formal mechanism for defining the interfaces between them. Thus a key requirement for a representation language is the ability to break down complex specifications into manageable units, which have well-defined interface properties. These *specification construction units* can then be reasoned about individually and subsequently combined in a formal way to produce a complete specification.

3.2.3 Justification for the Use of the Z notation

Whilst there have been many formal specification languages proposed, for this project it has been decided to use the Z notation (Spivey, 1992). The reasons for this are as follows:

- Z is a well-developed and accepted formal notation, which has been standardised. (Brien and Nicholls, 1992).
- The Z notation embodies powerful modelling abstractions that can be used to address a wide variety of requirements engineering problems.
- Z is based on set theory and first order predicate logic, which are well-understood mathematical formalisms.

- Most importantly, Z specifications are modular being constructed from abstract data types known as *Schemas*. These can be used as the building blocks of more complex specifications in conjunction with the *Z Schema calculus*.

Whilst the use of Z constrains the representation process to a model-based approach, Z has been shown to be applicable in a wide variety of problem domains including a number of industrial applications (Collins *et al.*, 1988). In addition, the ability of Z to modularise specifications into schemas is central to the specification process upon which the TranZit tool is based.

In terms of supporting the representation process, the requirements for a tool can be distilled to the need to be able to capture specifications written in the Z notation at the computer. Most users now expect some form of full-screen editor capability using a WYSIWYG GUI, driven by mouse and keyboard input. Given the conventions embodied in most Windows™ editor programs for example, users expect standard features to be available such as cut and paste, delete, select and insert, as would be found within a standard text editor.

However, in addition to these essential features, there is a need to add support for representational aspects of the chosen notation, in this case Z. At a simplistic level, many of the specialist mathematical Z characters are not available within standard computer character sets, and must therefore be made accessible to the user. Also, even the most experienced of Z writers can forget the notation symbols for various operators, and the tool must therefore support the efficient location of Z notation characters by functional group. The tool must also be capable of drawing schema outlines, and since the graphics associated with these objects are of no interest to the user, the tool itself should control these. To this extent, if the user wishes to enter further information within the predicates of a schema body, the body should expand accordingly without the need for the user to re-draw the schema graphic outline manually.

3.2.4 Product Requirements for a Tool to Support the Representation Process

From the preceding discussion, Table 2 identifies a list of high-level product requirements for a requirements engineering tool to maximise the efficiency of a representation process using the Z notation.

Representation Process Need	Tool Requirement
Ability to capture the specification efficiently	⇒ Full-screen, WYSIWYG editor combining mouse and keyboard input
Provide support for the chosen representation language	⇒ Support for Z notation character set ⇒ Automatic generation of notation graphics (e.g. schema outlines)
Ability to modify the specification efficiently	⇒ Support for standard editor functions: ⇒ Cut, paste, insert, delete, select, search and line goto, with appropriate notation support (i.e. cut, paste, delete of complete schemas)

Table 2: Product Requirements for a Tool to support the Representation Process

These requirements ensure that features are present in the tool which support the representational elements of a high quality Z specification. However, it is also necessary to identify how the tool will assist specifiers in refining Z specifications. This is the domain of the *specification process*.

3.2.5 Developing Requirements for a Tool to Support the Specification Process

Having chosen a formal notation for the representation process, it is necessary to consider how a requirements engineering tool should support the specification process itself. This involves support for the *specification refinement process*, beginning with a blank piece of paper and moving towards a complete set of requirements captured within the system specification.

There are several considerations involved in this process, as highlighted in the Figure 3-3:

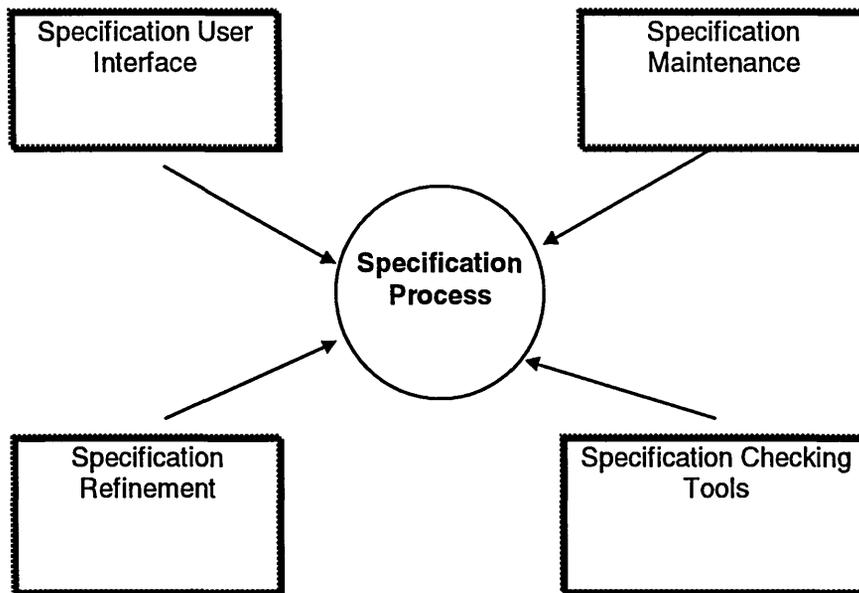


Figure 3-3: Considerations in the Specification Process

Of great importance is the way in which the tool presents the specification to the user. Regardless of the internal sophistication of any tool, if the user interface is inadequate the tool itself will be viewed as inadequate. One of the most popular and arguably the most successful of computer user interfaces available to date is the Microsoft Windows™ Graphical User Interface (GUI) (Microsoft Corp, 1992). Indeed, such is its proliferation throughout the desktop PC market that the interface is now readily available to everyday users on relatively cheap PC hardware.

Whilst other platforms such as the SUN OpenWindows GUI and the Apple Macintosh GUI were considered, none of these can compete with the Microsoft Windows GUI on the basis of general accessibility.

Most importantly, the Windows GUI imposes a consistent presentation method for Windows applications, in that the set of objects available to manipulate the application, i.e. *menu bars, scroll bars, buttons and controls*, are well understood by Windows users.

It is therefore often unnecessary for a user in this environment to begin learning each new application GUI from scratch, since all Windows applications offer a common interface style. On this basis, most experienced Windows users exposed to a new program can find their way around the basic manipulation tools very quickly, based on the conventions which have been adopted by Windows program designers.

From the preceding argument, the toolset presented in this project has been based on the Microsoft Windows platform, since it was felt that this environment offered the greatest accessibility for users and, most importantly, sufficient GUI flexibility to embody the design concepts of the proposed tool.

From the general considerations for requirements engineering tools discussed previously, the tool must recognise explicitly that the generation of a specification of requirements is a *creative* process. It is implicit within the creative process that some form of *refinement* is taking place, and that it is necessary to be able to manipulate, save and retrieve specifications and specification fragments independently as the specification task evolves.

In order to *maintain* the specification, there is a need to import or export information from the tool to other parts of the development environment. For example, developers may need to export specification fragments to a word processor in order to be able to generate supporting documentation for the project. Again the windows environment chosen supports well-defined mechanisms for achieving this. Similarly, the tool must be able to produce hard-copy of the specification for the purposes of discussion and review.

Finally, the tool must embody some form of support for the chosen specification notation and implement features to assist in the development of a specification in this notation. These features are different to the representation features such as standard editor facilities discussed previously, in that an *efficient* specification process is concerned with manipulating and refining elements of the specification whilst at the same time retaining internal consistency. However, to fully appreciate the issues associated with this problem, it is necessary to examine the process of developing a specification itself.

3.2.6 Understanding How Specifications are Developed

Whilst no definitive process exists for constructing Z notation specifications efficiently, several suggestions have been documented based on ideas developed by IBM Hursley (Wordsworth, 1987) and the Oxford University Programming Research Group (Wordsworth 1989). In general, these aim to offer a *proforma* for developing specifications in a consistent style, and are largely self-evident.

However, rather than be constrained by a prescriptive proforma, it was decided to use real-world problems to investigate the process of specification construction, using both introspective and observational techniques. The objective was to gain a better understanding of the specification development process, and thereby be able to refine the functions offered by the tool, to maximise its efficiency. This work was based on two large industrial projects within a major development organisation, concerning definition of user interfaces for desktop terminals.

In this organisation, the specification phase (or *definition phase* as it is sometimes referred to) is driven by two major *product-level* inputs:

- *The Marketing Requirements Specification (MRS)*
- *The Product Requirements Specification (PRS)*

The MRS is a high-level description of the perceived need for the product. It may outline the requirements for major features (functional requirements), but will not specify them in detail. In addition it may stipulate a number of Non-functional requirements (NFRs), relating to reliability, performance, interoperability and maintainability. The MRS will also stipulate schedules and cost targets, which will need to be met by the project. The document is written in natural language to a defined document structure. In conjunction with the business case, the MRS is a key component in assessing the feasibility of the product.

Once the product feasibility has been established the domain experts write the PRS in response to the MRS. The PRS is a detailed low-level specification of requirements for the system to be developed, written from the viewpoint of the major components or *complexes* which the system requirements demand. For example, in a telecommunications system, the PRS may stipulate *changes* to the PABX core software (which in itself would be defined as a *complex*), and would identify the need for *new* hardware or software components. It may also identify the requirements for peripheral equipment such as desktop terminals or PC Applications associated with the overall product. In addition, it will clarify NFR's associated with such factors as performance, MTBF and out-of-box quality.

Once the PRS is agreed by the stakeholders, domain specialist will become involved to develop a number of *Product Functional Specifications* (PFS) for the individual complexes identified by the PRS (e.g. system software changes, user interfaces for desktop equipment or new applications). The PFS documents are the definitive specifications from which the system designers will develop the required system. The process is summarised in Figure 3-4:

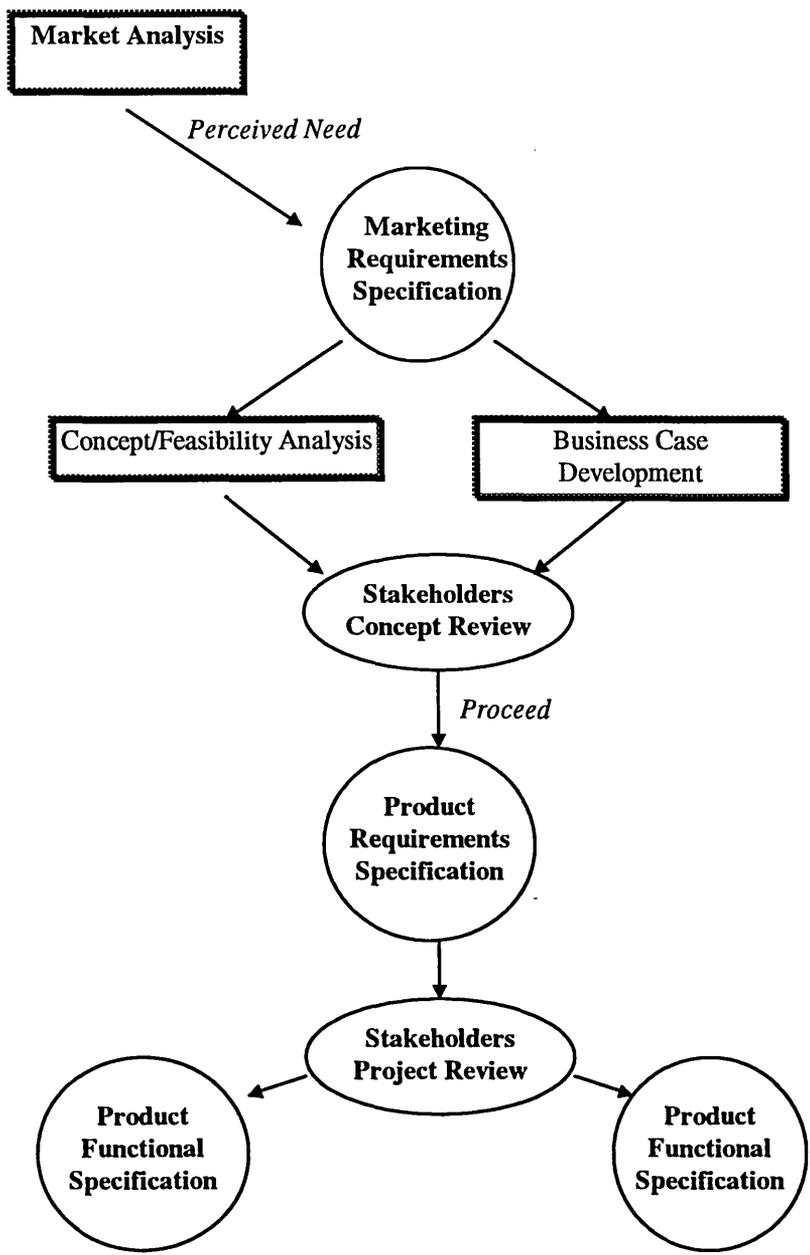


Figure 3-4: Industrial Specification Process

Whilst the processes involved in the definition phase are fairly straightforward, what is not clear is how the domain experts go about generating a specification from first principles, and the thought processes involved.

Whilst the organisation does not use a formal specification notation, experience with the DOORS tool (QSS Inc, 1998) for organising natural language requirements in an object-

oriented fashion, has identified a number of interesting points associated with writing the specification itself. Of particular interest are the *thought processes* involved, which appear to go through three main phases as described below:

3.2.7 The Usability Phase

On asking domain experts how they go about producing a specification from first principles, most will explain that they start by thinking about the major objects or *complexes* which make up the system to be specified. These objects are initially described not by their internal interactions, by how they interact with the *system environment*. Thus, in the case of a desktop terminal, the domain expert first considers the keyboard, the display and the audio interface as individual components and specifies how these should interact with the system environment, i.e. the user. This *usability phase*, seems to be a key phase in focusing user interface ideas, upon which many of the later decisions concerning detailed functional requirements will be based. Indeed it is not uncommon for a usability prototype to be generated at this stage which models the interface to the system environment, without concern for the detailed features to be offered. The usability phase also seems to highlight the major system constraints, from which the domain expert will ultimately generate a set of NFRs.

By this stage, the specification will be a skeleton document that describes:

- The required layout of the display/user interface (usually diagrammatically),
- The required layout of the features buttons/menus on the GUI or user interface,
- A high-level description of system features,
- A description of interface requirements to the external system environment,

At this stage a usability review of the specification is normally be held, to ensure that all stakeholders agree with the major objects being included in the specification, and to ensure that no gross functionality or interfaces have been overlooked.

3.2.8 The Functional Phase

Following the usability phase, the domain expert then focuses mainly on the specifying the *features* to be implemented as functional requirements. Interestingly, this is normally seen as a serial task, with each feature being described in turn, without reference to existing definitions. The important task of resolving *feature interactions* is deferred until the end of this phase, or more often the *resolution phase* described later. The aim of this *functional phase* is to set in place how the features of the system will operate and modify the states of the system objects defined in the *usability phase*. This ultimately generates relationships between these objects, which in turn define their interfaces. A common tool used to support this phase is the generation of an Entity Relationship diagram (ERD).

This phase tends to take the most time, since it is at this point that the detailed requirements for individual features and functions are defined. The specifier's aim at this phase seems to be to produce as much detail as possible concerning individual features, however the specifier seems less concerned with how these features will interact. This appears to be the case even when there are several specifiers working on the same specification.

3.2.9 The Resolution Phase

Once all the functional requirements are completed, a detailed review is normally held, involving all member of the stakeholder team. This review has two main objectives:

- To ensure that all functional requirements are specified in sufficient detail
- To highlight interactions between functional requirements which need resolving

Until this point, the specifier(s) will have made only a passing attempt at resolving *interactions* between features. This is normally deferred to this review when all members of the stakeholder team can bring their own expert knowledge to the discussion. Interestingly, it was observed that at this review the expert specifier is often content to take a passive role, and let other members of the stakeholder team identify inconsistencies and propose solutions. There are perhaps several reasons for this:

- The expert specifier feels that he will have discovered all the inconsistencies he is likely to discover in the functional phase, and needs the assistance of a fresh point of view.
- The expert specifier feels that by this point he has become too close to the system functionality, and the system needs to be re-assessed from a higher level.
- The expert specifier wishes to devolve ownership of the specification back to the stakeholder team.

This review is by far the most intensive, and as it progresses it may become quite heated as individual stakeholders attempt to impose their viewpoint on the system specification. A strong chairman is vital, and in addition Marketing authority must also be present to make decisions concerning trade-offs between conflicting requirements. If necessary, ranking of requirements by the Marketing authority will also be undertaken, where project schedules are unlikely to be achieved given the complete set of requirements.

The ultimate result of this review may have several outcomes.

1. The stakeholder team agrees that the PFS's produced are a true representation of the system requirements. The PFS's are then approved as the primary input to the development team to begin implementation. It is noted that this outcome is unlikely in the first pass of the specification review, for anything other than a trivial project.
2. The stakeholder team determines that either functionality is missing, is badly defined or that conflicts cannot be resolved without rework to the functional aspects of the specification. Another review will then be scheduled once the requirements have been refined by the domain expert(s). In general it is found that at least one repeat review is required.
3. The stakeholder team determines that a major system component or interface is missing, and it is necessary to revisit the usability phase of the project. In general this outcome should not occur, as it is indicative of poor communication between the stakeholders and a lack of understanding of the high-level product requirements.

3.2.10 Conclusions from the Study of Specification Development

The research into the development of specifications has highlighted a three phase iterative model consisting of the usability phase, the functional phase and the resolution phase, as shown in Figure 3-5:

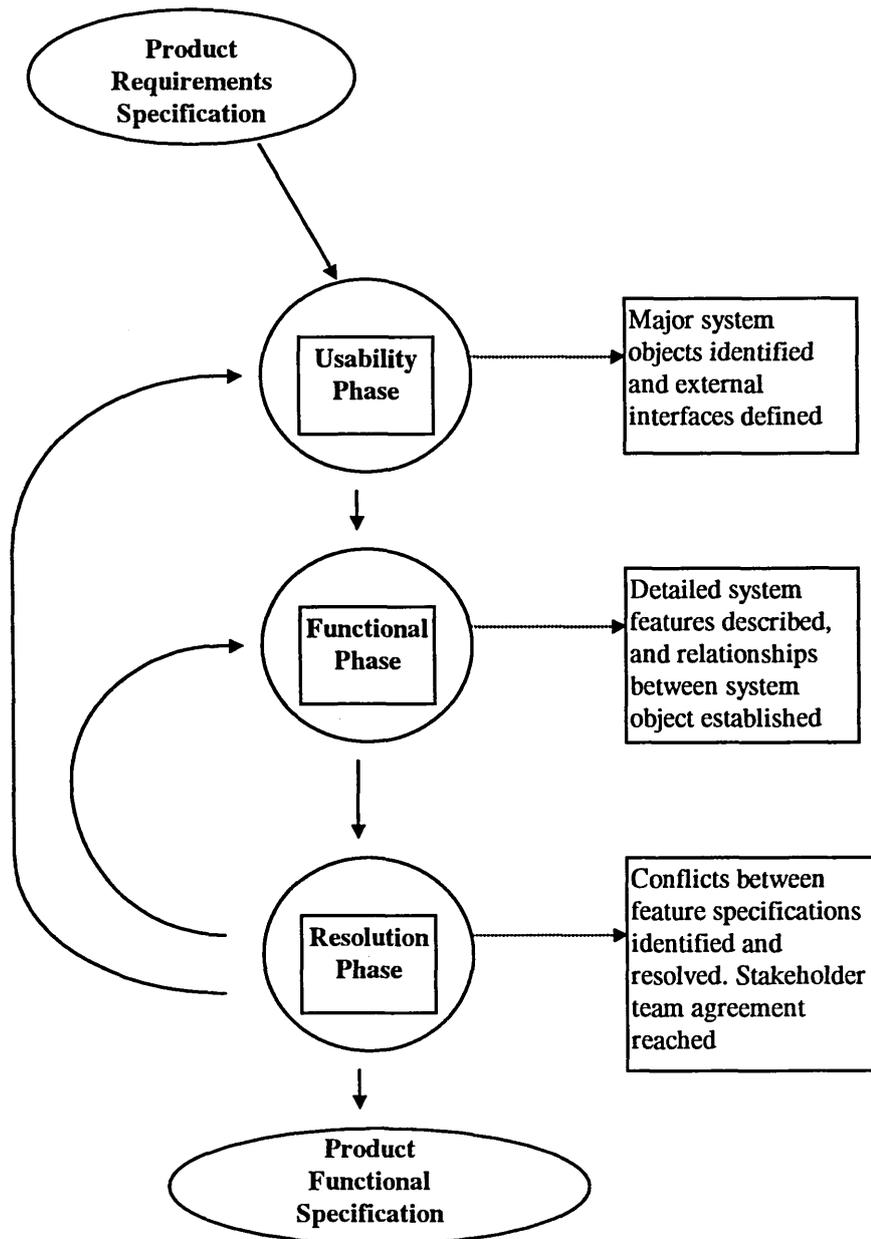


Figure 3-5: The three phases of Specification Production

Having understood how these phases come about, it is now possible to consider how a tool should go about supporting these phases in an efficient manner.

The specification itself is considered to consist of a number of descriptive abstract specification objects (i.e. natural language statements or formal descriptions of schemas in Z, for example). The development of a specification is the process of manipulating these objects to form a complete description, accepting that:

- *It is natural that the specification will change throughout its construction, and that the process of its development is inherently iterative and refining.*
- *The specification is complete when these objects are related in such a way as to produce a consistent and complete statement of requirements upon which all stakeholders can agree.*

Accepting that the specification process is inherently creative, each of the specification objects can be in a particular state, either:

- *Created (i.e. known to be needed but not yet defined in detail),*
- *Evolving (i.e. in the process of detailed definition), or*
- *Finalised (i.e. a complete object whose internal and external behaviour is defined in sufficient detail as required by the specification).*

The essence of a tool to maximise the efficiency of the specification process is therefore the ability to manipulate and maintain consistency between these evolving specification construction objects.

3.2.11 Managing the Interfaces Between Specification Components

From this research it is believed that one of the main problems with specification construction is managing the *interfaces* between the individual *complexes* or specification objects. When these interfaces are confused or too complex, problems such as ambiguity, redundancy and opaque understanding begin to occur.

It is clear from the investigation that specifications evolve in much the same way as software designs evolve, since this is the most natural way to break down and assimilate problems. Thus, in much the same way as a design tool, a tool to support the specification process must manage the *interfaces* between specification objects and support the evolution of these objects through their individual lifecycles in an organised and methodical way.

The purpose of a tool supporting the specification process is therefore to reduce the workload of the specifier, associated with maintaining the interfaces between the specification objects in a consistent fashion, and to support the use and manipulation of the chosen specification language (in this case the Z Notation).

3.2.12 Assessing the Impact of Formal Methods on the Specification Process

It has already been deduced that the major work associated with developing a specification involves maintaining the *interfaces* between individual specification construction objects. Yet the use of a formal notation such as Z imposes additional considerations in achieving this goal.

To understand where the work associated with maintaining these interfaces arises, it is instructive to use the analogy of *cohesion and coupling* from structured design methodologies (Sommerville, 1985). In a similar way that these design criteria affect the workload associated with maintaining a program, they also provide a framework to assess the work impact for a given change to the system specification. Clearly, where a modification has a large work impact on (or is *tightly coupled to*) the existing specification objects, this is where a tool should attempt to minimise the effort required to maintain the specification in a consistent state.

Within the TranZit tool, the specification construction objects are components of the Z notation (e.g. given sets, schemas, predicates, global variables, e.t.c.). Using experience in developing Z specification and applying the three-phase specification development

model discussed previously, leads to the following conclusions concerning the development of formal specifications in the Z notation.

- The *usability phase* associated with developing a Z specification is concerned with identifying the given sets and axiomatic definitions required to model the problem. This process is likely to be highly iterative, as the specifier is attempting to establish a coherent set of specification construction objects representing the external view of the system. However, as this iteration within the usability phase progresses, the introduction of new given sets and axiomatic definitions tends to have little impact on what has gone before. This is because, given sets invariably represent the introduction of a complete object (or complex) within the specification, which is likely to have its own functionality and requirements (i.e. it is a self-contained problem, with little impact on existing requirements). Similarly, axiomatic definitions place constraints or limits on the overall specification, rather than affecting individual components. Hence it is suggested that given sets and axiomatic definitions are *not tightly coupled* to the specification process, and a tool need not place much emphasis on assisting the user to maintain them.
- Whilst the given sets and axiomatic definitions identified within the usability phase are generally independent objects, the development of *general theories* captured as *system invariants* are more tightly coupled to the specification process. These system invariants are represented as *schema* objects, and represent major decisions concerning constraints on critical components of the system state. Hence any modification to these schemas once the *functional phase* has begun is likely to have a large effect on the work completed so far. For example, consider the effect of changing the declarations section of some schema representing a system invariant, which is in turn included by many other schemas. General theories and associated schema definitions are therefore *tightly coupled* to the specification operations, and a tool must assist the user in maintaining them throughout the specification process.
- During the *Functional phase*, components are introduced to the specification that model operations on the system state. We introduce functions, relations and abstract

data types to represent the *system state*, which in turn build relationships between the given sets identified as basic components of the model in the usability phase. In order to define *operations* we introduce *schemas* that collect together a set of predicates, which in turn define a mathematical description of changes to these elements representing the system state. The structure of the elements representing the system state may therefore permeate many schemas, as each must operate on the system state to perform some operation. Changes to the *type* of a function for example, may therefore invalidate many operations using that function. The elements of the system state are therefore *tightly coupled* to the specification as one might expect, and may undergo many modifications as new features are modelled and the interactions between features are resolved. The tool therefore needs to support the user in checking the *internal consistency* of the elements representing the system state as the functional phase progresses.

- In the *Resolution Phase*, the emphasis is on resolving the interactions between operations represented as schemas. The work associated with achieving this will be heavily influenced by the *cohesiveness* of the way in which schemas have been written. In the same way as it is possible to write badly structured programs, it is equally possible to write badly structured specifications in Z, which increase the maintenance effort required. For example, if *schema inclusion* has not been used effectively, the scope of any modifications required may be quite widespread. It therefore follows that the tool must support the schema inclusion and schema hiding semantics of the Z notation.

3.2.13 Product Requirements for a Tool to Support the Specification Process

From the preceding discussion, it is now possible to draw up a list of product requirements for a tool to maximise the efficiency of the specification process using a formal notation such as Z. The list is shown in Table 3.

Specification Process Needs	Tool Requirements
Provide support for evolutionary development and refinement of the specification	⇒ Ability to load and save work to hard disk or floppy drive. ⇒ Ability to print the specification on standard printers
Provide support for exchanging information with other components in the development environment	⇒ Support interworking with other specification documentation packages (e.g. MS Word).
Support the process of specification development by refinement of specification units	⇒ Provide automated support for the generation of schema components within the specification
Support the generation of concise and cohesive specifications	⇒ Provide automated support for the semantics of schema inclusion and schema hiding
Support the evolution of the abstract state model.	⇒ Provide automated support for type generation and checking within the Z notation specification.
Support the functional modelling process to ensure internal model consistency	⇒ Provide Automated tools for syntax checking of Z notation specification

Table 3: Product Requirements for a Tool to support the Specification Process

It is interesting to note that these requirements call for some form of internal consistency check (syntax or semantic analysis) within the tool. Several tools of this type, e.g. *Formaliser* (Logica Inc, 1995) offer *syntax-directed editing*, in which the tool only allows syntactically correct constructs to be entered as defined by the specification language parse tree. The alternative approach is akin to the conventional compiler, in which an off-line syntax checker is provided as a separately invoked function.

The research into specification writing in Z revealed an interesting point on this subject in that, although Z specifications have formally defined syntax, *getting the syntax correct is not the primary concern during the initial usability phase of the specification process*. Rather, at this point the specifier is more concerned with capturing the essence of the model than syntactical correctness, and may find that the need to concentrate on complicated syntax is a source of distraction. Hence to support this finding and to enhance the creative processes involved in the early phases of the specification process, it was decided to provide an off-line syntax checker, rather than enforce a syntax-directed editor on the specifier.

A similar argument applies to the type checker. It is noted that many users find the Z type system the most difficult part of the notation to understand. Hence, most users tend to defer this problem until later stages of the specification process, when the model concepts are more firmly defined. Moreover, users can usually identify syntax errors more readily than type errors, and hence these tend to get resolved first. Therefore, the tool should not only provide a separate syntax checking phase, but the user may select whether this check is to include type checking or not. In this way, the user may take the process of generating internal consistency within the specification in easy stages.

3.2.14 Developing Requirements for a Tool to Support the Agreement Process.

As discussed previously, each stakeholder involved in the requirements engineering task will present a different set of concerns and viewpoints. The requirements engineering process must foster a common understanding of the system specification, and in addition must foster an environment within which agreement on its contents can be reached. Without this *agreement process* the requirements engineering task can never be concluded to the satisfaction of all stakeholders.

Whilst the representation and specification processes will assist in the production of a high quality specification, they cannot prove that the specification is a true reflection of what the stakeholders actually want. Secondly, they cannot prove that the specification can be implemented by a computer system. This leads to two key problems in

requirements engineering which will be termed *the semantic gap* and *the implementation gap*.

3.2.15 The Semantic and Implementation Gaps

But this book cannot be understood unless one first learns to comprehend the language and interpret the characters in which it is written. It is written in the language of mathematics ... without which it is humanly impossible to understand a single word of it.

Galileo Galilei

Il Saggiatore

The specification is a contract between the stakeholders and the developers. The use of mathematical formalism helps to make this contract more precise, but it does not address the crucial issue central to the agreement process, that of *communication*. Indeed, since the Z notation used in the specification process is of an abstract mathematical nature, it is unlikely to be understood by non-technical members of the stakeholder teams. Therefore, how can agreement be reached if the stakeholders cannot understand what is being proposed? This is a *crucial limitation* brought about by the mathematical nature of formal specifications.

In addition, formalism can be used to *verify* that an implementation meets the specification (Gries, 1981), but it cannot show that the specification can be implemented on a physical computer. This problem arises from the gap between the *abstract* nature of the objects used in the specification model, and the *concrete* structures available to computer programmers. For example, the set objects involved in modelling a specification may be legitimately infinite in size, however a computer does not have infinite memory and cannot feasibly complete a search of an infinite object in finite time. It is also the case that the types of objects used may be assumed to have infinite precision (e.g. real numbers), but may be constrained to the size of the machine word in a physical implementation.

Hence, to support the agreement process, a tool must support and foster a common understanding of the meaning of the captured specification, at a level which all

members of the stakeholder team (both technical and non-technical) can readily assimilate. The achievement of this objective is termed the common view. Secondly, the tool must give some confidence that real world objects such as computer software and hardware can implement the specification objects captured.

As discussed previously, this project addresses the first problem by the use of *Animation of specifications*. This process involves the provision of an environment in which the user can populate the state space represented by the specification model with candidate data. The user may then invoke operations from the specification on the candidate data, and observe the changes to the state space. If the behaviour is as expected, then there is a fair degree of confidence in the fact that the specification meets the requirements of the user. In addition, the user may also be able to deduce additional properties of the specification by animating “*what-if*” scenarios.

Clearly, this animation process requires that the specification itself be transformed from the non-executable Z notation into some language that can be *executed* by the animation system. Since it is pointless for the users to do this by hand (since they would be in effect building the required system), we expect that a tool should be capable of performing this task. The executable form of the specification is essentially a *rapid prototype*, since it generated automatically by the tool on demand and it is a skeletal representation of the objects defined in the original specification. Hence the ability to *transform* a formal specification into an executable representation is a key requirement for the tool. If this can be achieved, then we also implicitly address the *implementation gap* by showing that we can implement the specification as a rapid prototype in computer software.

3.2.16 Product Requirements for a Tool to Support the Agreement Process

From the preceding discussion, it is now possible to draw up a list of product requirements for a tool to support the agreement process using a formal notation such as Z. The list is shown in Table 4.

Agreement Process Needs	Tool Requirement
<ul style="list-style-type: none">• Provide a mechanism for communicating the meaning of the specification in terms that all member of the stakeholder team can understand.• Provide a mechanism for ensuring that the specification can be implemented by physical objects (computer hardware and software).	⇒ Ability to transform the specification from its non-executable form, to an executable representation, suitable for validation in an animation environment.

Table 4: Product Requirements for a Tool to Support the Agreement Process

3.3 A Process for Requirements Engineering Using TranZit, ZAL and ViZ

Thus far, we have investigated the requirements for a requirements engineering toolset to address the generic representation, specification and agreement processes, and from this work produced a set of high-level product requirements based on research into the behaviour of stakeholder teams and the mechanisms involved in Z specification construction.

Based on this research a toolset has been defined which aims to support a requirements engineering process based on the *capture* of formal specifications written in the Z notation and specification *validation by execution*. As discussed previously, the toolset does not attempt to assist the *elicitation* process, since this process focuses on human communication techniques that are difficult to support by computer-based tools.

The toolset associated with this project consists of three integrated, complementary tools, which are loosely coupled to provide a complete requirements engineering workbench based on the Microsoft Windows™ Operating System.

The requirements defined previously to support the representation and specification processes are implemented by a tool called *TranZit*. *TranZit* is a Windows-based requirements engineering tool for capturing Z specifications, and automating their transformation to an executable representation. It incorporates powerful features supporting the construction, manipulation and maintenance of Z specifications, as well as tools for checking the internal consistency of Z specifications including a complete Syntax Analyser and Type Checker. In addition, *TranZit* incorporates a novel *Transformation Engine*, which allows captured specifications to be automatically transformed (as far as possible) into an executable representation, suitable for input to an associated animation environment called *ZAL* (Z Animator in LISP).

In addition, the animation environment is supported by a tool known as *ViZ* (Visualisation in Z), which allows *graphical representations* of specifications to be animated. This level of animation is at a higher level than *ZAL* and allows stakeholders to view the operation of the system, using icons and graphics representing the real-world objects modelled in the specification.

The agreement process is supported by *TranZit*, *ZAL* and *ViZ* as a co-operating, integrated animation environment. The *ZAL* tool provides the execution engine for the animation environment, accepting Z specifications that have been transformed into the *ZAL* language by *TranZit*. *ZAL* provides a user interface to allow stakeholders to populate the state space with candidate data, and investigate the properties of the specification by use cases. This process is termed *validation by execution*.

Whilst reference is made to *ZAL* and *ViZ* in what follows, **it is the research and development of the *TranZit* tool which forms the basis of the work presented in this thesis.** A complete description of *ZAL* and *ViZ* is beyond the scope of this thesis, and in what follows they shall be treated as separate entities to *TranZit* that export an interface

accepting transformed Z specifications in the ZAL language. In this way, *TranZit* need only know the grammar of the ZAL language and the exported communication methods to be able to interwork with the ZAL and ViZ applications. This has the additional benefit that *TranZit* is independent of ZAL and ViZ and can be used as a stand-alone tool in its own right. For further details of ZAL see Morrey *et al.* (1998) and Siddiqi *et al.* (1997). For further details of ViZ see Parry *et al.* (1995).

3.3.1 The REALiZE Process

The REALiZE process (Requirements Engineering by Animating LISP incorporating Z Extensions), has been developed to formalise the interplay between requirements acquisition, requirements formalisation and requirements validation, as embodied by the *TranZit*, *ZAL* and *ViZ* toolset.

The process fits into the standard software lifecycle model at the requirements analysis phase as shown in Figure 3-6:

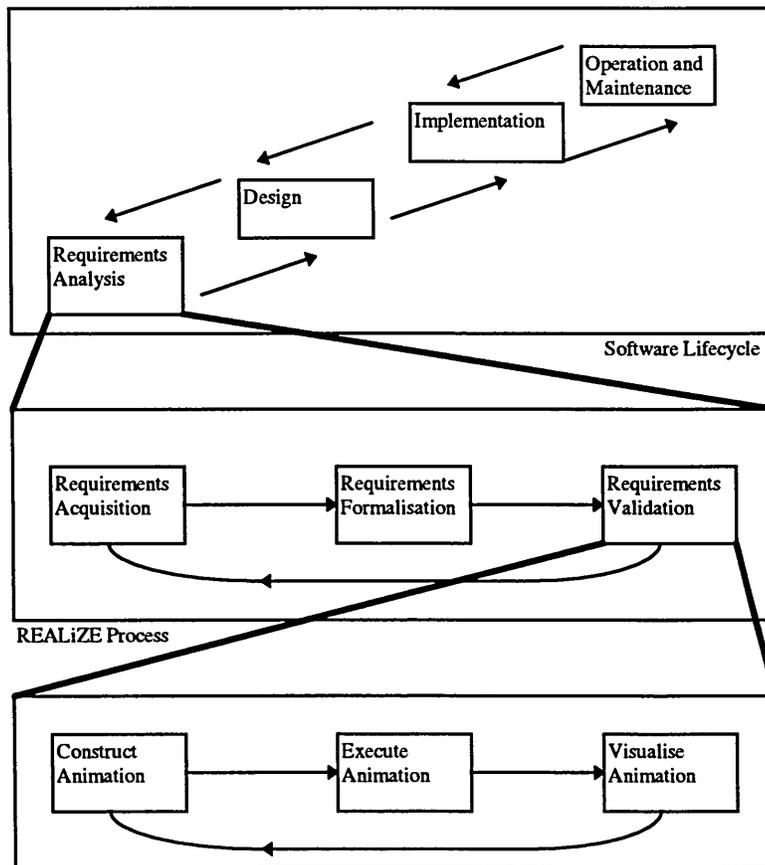


Figure 3-6: The REALiZE Process

Following an initial *requirements acquisition phase* involving techniques such as interviewing domain specialists and user questionnaires, the specifier enters the *requirements formalisation* phase. In this phase, the requirements are captured by the

specifier in the Z notation using the facilities provided by the *TranZit* tool. Once the specifier is content that the formalisation is complete and that the specification captured in *TranZit* is the best representation of the requirements possible at this stage, then the specifier enters *the requirements validation phase*.

In this phase, the specifier first uses the *transformation engine* built into the *TranZit* tool, to produce an executable representation of the captured Z specification in the ZAL language (based on extensions to LISP). The *TranZit* tool then forwards this executable representation to the ZAL environment. This representation can then be *executed* by the specifier within the ZAL animation environment, for the purposes of demonstrating properties of the captured specification to members of the stakeholder team. This can be achieved by a number of methods:

- *Scenario Walkthrough*, in which use-cases are investigated representing the normal operation of the system.
- *Provocative Investigation*, in which attempts are made to make the specification *fail* to exhibit some desired property.
- *Exploratory Investigation*, in which “*what if*” scenarios can be proposed and investigated.

This process can take the form of either a formal review in which all stakeholders participate, or simply at a peer review level. The aim is two-fold: Firstly to clarify understanding of the specification itself for the benefit of all stakeholders, and secondly to improve the quality of the specification by ensuring that the requirements embodied are a true representation of what the system needs to do.

Finally, in order to make the system specification accessible to others outside the direct stakeholder team, who may not possess detailed knowledge of the system proposal, the *ViZ* visualisation tool can be used to produce a graphical representation of the required system. This further enforces the validation process.

The logical interfaces between the individual tool components associated with the REALiZE process are shown in Figure 3-7:

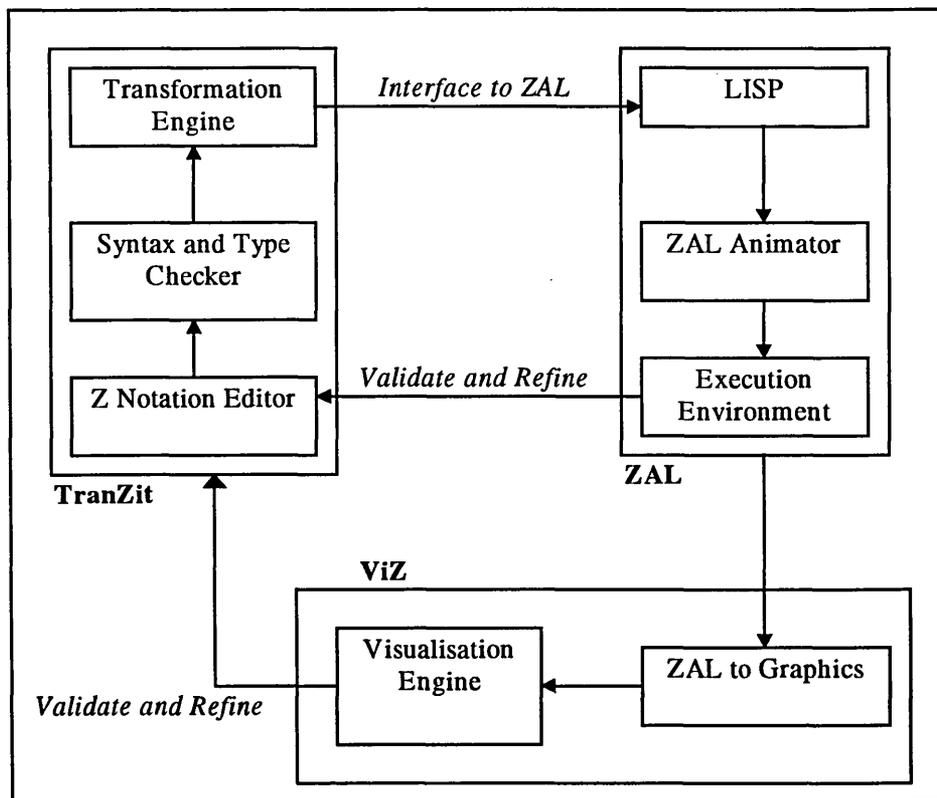


Figure 3-7: The Logical Interfaces between TranZit, ZAL and ViZ

It can be seen that the essence of the REALiZE process involves an *iterative* cycle of *validation* and *refinement* of the captured Z specification using *TranZit*, *ZAL* and *ViZ* in a co-operating environment. In this way the process retains the benefits of capturing the specification in the formal Z notation, whilst at the same time offering an environment which fosters effective communication between all members of the stakeholder team. Thus the aim of the toolset is to maximise the benefits of the techniques chosen, in order to produce a set of quality requirements embodied in a formal notation.

3.4 Summary

This chapter has explored the need for computer-based requirements engineering tools, and developed a taxonomy of tools by analysing the needs of the stakeholders and the

viewpoint they present in the requirements engineering task. This research has been used to develop the three primary uses for requirements engineering tools as:

- Tools for capturing and validating system functionality,
- Tools for managing conflicts between requirements and resources
- Tools for improving system reliability and maintainability

From these initial groups, a requirements engineering tools hierarchy has been developed which identifies where tools can assist in the requirements engineering task.

Using Pohl's model of the component processes within the requirements engineering task, a set of high-level product requirements for a requirements engineering tool based on the use of the Z notation have been developed. These requirements have been based on research into the process of specification construction itself, and the thought processes used by domain specialists in developing specifications in an industrial environment.

Finally, the development of the REALiZE process has been discussed, which is supported by a toolset consisting of *TranZit*, *ZAL* and *ViZ*. Together, these tools form a powerful integrated environment supporting the capture, animation and visualisation of formal specifications written in the Z notation.

The remainder of this thesis focuses on the research and development of the *TranZit* tool. The next chapter begins this process by exploring the detailed design and implementation of the *TranZit* Editor and Analyser subsystems.

4. Realisation of the TranZit Editor and Analyser Subsystem

This chapter discusses the research and development of the TranZit editor and the TranZit Analyser Subsystem (TAS). *TranZit* provides the user front-end to the integrated REALiZE toolset, as well as supporting checking and transformation of the captured specification for use in animation. As well as being a member of the REALiZE toolset, *TranZit* is a sophisticated requirements engineering tool in its own right, addressing many issues associated with the representation and validation phases of the generic requirements engineering lifecycle.

The main features of *TranZit* are as follows:

- *TranZit* includes a powerful, full-screen Z editor that presents a WYSIWYG GUI in which the user can construct Z notation specifications from user requirements. The editor is *language-aware*, allowing it to automate many of the formatting and specialisations required in the use of the Z notation, without constraining the creative process of specification development itself.
- *TranZit* includes a complete Z notation syntax analyser based on an optimised version of Spivey's (1992) original grammar, which has also been extended to meet the requirements of the Z base standard V1.0 (Brien and Nicholls, 1992).
- *TranZit* includes a complete Z notation type checker, again derived from Spivey's language definition.
- Most importantly, *TranZit* includes a *novel transformation engine*, which has been designed to automate (as far as is possible) the process of converting the Z notation specification into a procedural representation in the ZAL Language. It is this feature which provides integration with the other animation tools in the REALiZE process.

In this chapter it will be shown how existing graphical user interface and compiler design techniques have been combined with research into the requirements engineering representation and validation processes, to develop a tool which assists the specifier in producing a high quality specification from *ad-hoc* user requirements.

The research and development of the TranZit Analyser Subsystem (TAS) is also described, which is a highly efficient syntax and type checker for Z. The TAS makes use of traditional compiler design techniques coupled with innovative research into object-oriented data structures to support internal consistency checking of Z notation specifications.

4.1 Research and Development of the TranZit User Interface

The interface that a program presents to the user is perhaps the most important part of any professional development tool, or indeed any program in general. The reason for designing a computer program to perform any task is to save time and effort. Hence the aim of the user interface is to ensure that the effort required to enter data into the program, manipulate it within the program, and collect results from the program is performed in the most effective manner possible.

As highlighted by Thimbleby (1990), user interface design is a very difficult business as it combines two awkward disciplines; psychology and computer science. These disciplines have very different cultural backgrounds: Psychology is concerned with understanding people, whilst computer science is concerned with understanding computer machinery. Good user interface design therefore requires that both these perspectives be considered.

With many early programs, users were often faced with the intellectual challenge of having to work out how to make the program accept information, and then how to manipulate that information, even before any results could be obtained. This often led to user frustration and dissatisfaction, overshadowing the usefulness of the program itself.

A good example of this problem is the Vi editor tool, which is a standard component of the UNIX[®] operating system. In itself, this is an excellent full-screen editor with many powerful features and a long-established reputation for reliability. However, the user interface of the program is so complex as to be very difficult to learn, requiring multiple combinations of key presses to perform tasks, and knowledge of special command line syntax. To users who have taken the time and effort to learn Vi, the full power of the tool is readily available to them and the majority hold the tool in high regard. However

the learning curve involved in acquiring the skills to use the tool in the most efficient manner, leaves the full power of the tool inaccessible to many would-be users.

This type of problem is typical of many programs designed by domain specialists, in which the designer focuses on the internal algorithmic complexity dictated by the features that the tool must provide, to the detriment of the user interface design. The result is often a very powerful program with a well-defined set of facilities, but which can be very difficult to use.

Some would argue that it is the quality of the user documentation that ultimately dictates the usability of a program. However, experience suggests that the majority of users, especially technically-minded users, will begin by *triallying* the program by experimentation. This is an attempt to get a *feel* for the program before resorting to the user documentation for a detailed explanation of its operation. This *trial* phase is often critical in colouring the user's perception of the program, and ultimately influencing whether they intend to continue to use it.

Hence it is important that as many of the program's facilities as possible are *intuitively* available to the user, without needing to address the documentation reference. Whilst the quality of the documentation and training associated with a program are very important, many users view the ease with which they can immediately begin getting results from a program as a benchmark of whether they will continue to use the program. Indeed it could be said that the essence of the usability problem, or the achievement of a *user-friendly* program, is mainly concerned with building *intuition* into the user interface.

In general, the principles of a good user interface design include:

- Well laid-out and clearly presented screen designs, making *appropriate* use of graphics to enhance presentation, delineate features and improve understanding.
- A well thought-out, intuitive input mechanism (utilising common keyboard characters, commands or mouse input).
- A well-defined boundary between input mechanisms which control the way the program operates and the entry of program data (this is one of the reasons why the

use of a keyboard and mouse is such a powerful input combination. Users associate the mouse with program control and the keyboard with data entry).

- Flexibility in the way in which data can be entered.
- Tolerance to user uncertainties and mistakes.
- Clear use of language and symbols to identify features.
- Appropriate user-guidance by the program.
- Appropriate and clearly understood error reporting.
- Readable and comprehensible output, presented at a rate which can be readily assimilated by the user.

The amount of effort required to engineer a well-designed user interface should not be underestimated, and can often contribute a large amount of intellectual prototype work as well as detailed design time to the project. However, since this is the only part of the program that the user will ever interact with, it is vitally important that the right level of abstraction is attained between the program function and the user interface it presents.

A complete discussion of user interface principles is beyond the scope of this work, and the goals of user interface development have been well researched. For an excellent introduction to the subject see Thimbleby (1990).

The majority of popular user interfaces, such as the Microsoft Windows™ user interface, employ some form of graphical user interface (GUI) to increase the expressive power of the program by utilising screen designs and icons easily assimilated by the user. Indeed, this style of user interface now dominates the PC market due to the proliferation of the Windows operating system. The remainder of this section describes how features of the Windows user interface have been used to present the GUI of the *TranZit* tool, associated with capturing and manipulating Z notation specifications.

4.1.1 The TranZit User Interface Design

The *TranZit* screen design is based on the principles of the Microsoft Windows™ GUI. This is a well-established, well-understood interface, which embodies particular conventions that experienced users expect to find as part of a Windows application. As

far as possible, these conventions have been observed in the screen design of *TranZit*, allowing experienced Windows users to begin using the program with the minimum of training.

The philosophy behind the TranZit user interface is to offer the user a comprehensive set of tools for capturing and manipulating Z specifications, and to be able to use the captured specification in the wider context of animation. However, on a practical level, the tool is intended for use by both a novice user unfamiliar with Z, and also an experienced requirements engineer.

To accommodate these goals, the TranZit user interface is designed to present the features of the tool in a hierarchical fashion based on the experience level of the user, as show in Figure 4-1:

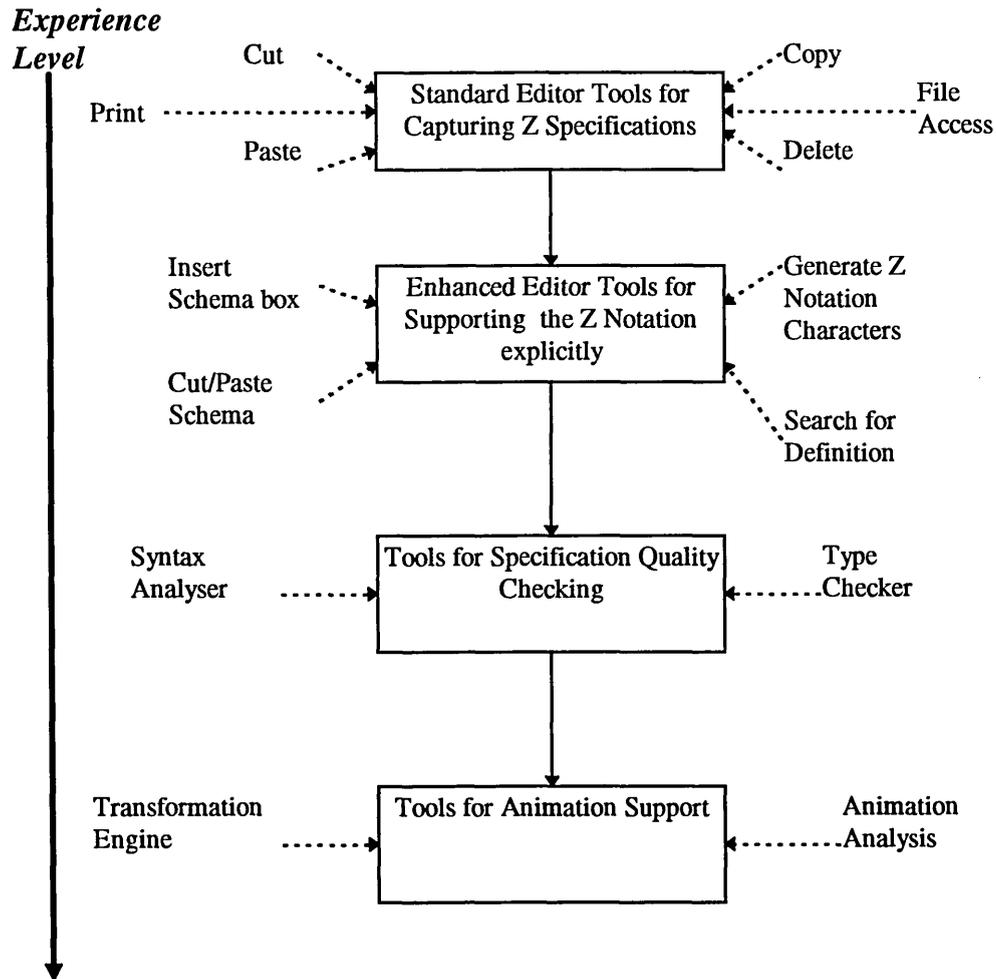


Figure 4-1: Hierarchy of TranZit Features

The aim of this approach is firstly to accommodate different levels of user experience with the Z notation, and secondly to provide a *learning vehicle* to increase the user's understanding and experience of formal methods through using the tool itself.

4.1.2 The TranZit Main Editor Window

On executing, *TranZit* presents the main *editor window*, which consists of a *Menu bar* from which features are selected, and a *client area* where the specification itself is constructed. The main editor window is shown in Figure 4-2:

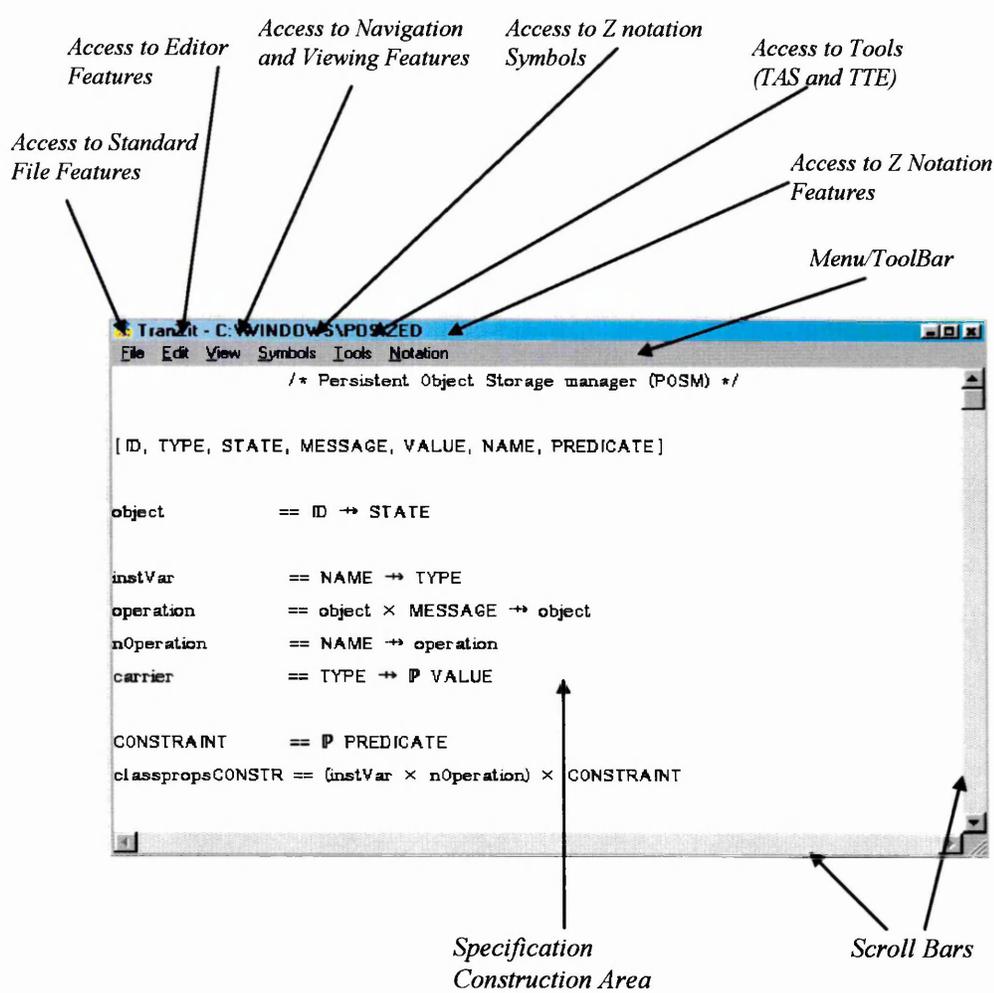


Figure 4-2: TranZit Main Editor Window

At this level the TranZit user interface appears much the same as any other Windows application, and this is a deliberate design aim to ensure that the tool adopts the conventions of the environment in which it is intended to operate.

The functions of the tool are grouped on the menu bar according to standard Windows conventions. For example, under the *FILE* menu are tools to load, save and print the contents of the specification. Under the *EDIT* menu are tools to manipulate the contents of the specification during an editing session. Similarly, under the *VIEW* menu are tools to navigate around the captured specification efficiently. However in addition the system presents specialist menus accessing functions to manipulate Z specifications specifically such as the *NOTATION*, *TOOLS* and *SYMBOLS* menu.

4.1.3 The Use of Object-Orientation to Support the Capture of Z specifications

Whilst one would expect *TranZit* to include a basic set of editing tools (e.g. cut and paste), *TranZit* is much more than a simple editor system and incorporates enhanced Z *language-supporting* features within the user interface design.

Normally, basic editors deal with simple character objects. However, to achieve the language-supporting features, additional *specification construction* objects are made known to the system, which can be used by both the editor, syntax analyser, type checker and transformation system.

In particular, *TranZit* treats schemas as *objects* in their own right. The object is created when a schema is created in the editor, and additional attributes are then added as more sophisticated tools are employed (i.e. types are added by the type checker). The editor recognises three derived classes of the virtual base class *Schema*, as in *Schema box*, *Generic Schema* and *Axiomatic Schema*. The editor has knowledge of how to create the graphical outline of these objects, and can manipulate them as individual editor objects. The editor is responsible for maintaining the integrity of these objects within the system, and provides methods to manipulate them. For example, the only way to create a schema box in *TranZit* is to use the *OPEN_SCHEMA* tool from the *NOTATION* menu bar. The user is then presented with a dialog in which to enter the attributes of the required schema object, as shown in Figure 4-3.

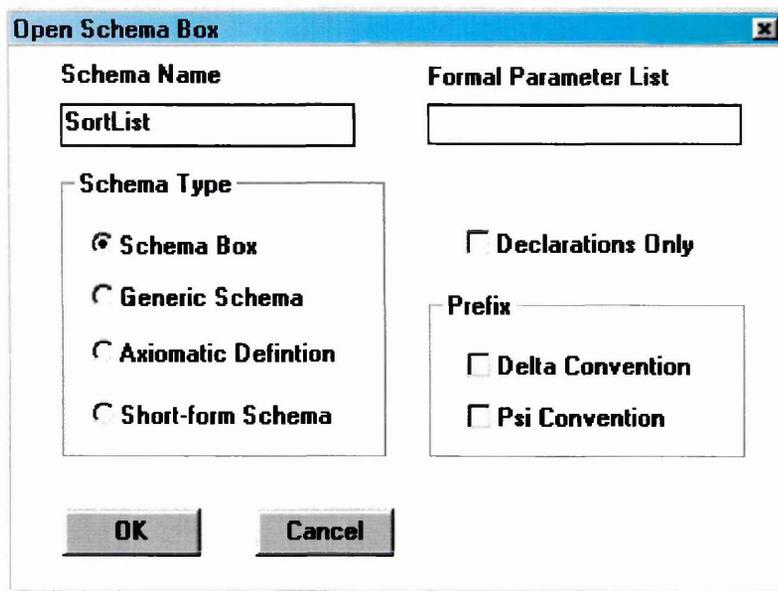


Figure 4-3: TranZit Open Schema Dialog

Once the relevant parameters are entered, *TranZit* will create, draw and maintain the integrity of this schema object internally. In particular, the editor will automatically expand the schema graphical outline as information is inserted into the declarations and predicates sections. This allows *TranZit* to help the user to maintain the specification in a consistent state, as the user cannot access the attributes of the schema object directly (for example, the user cannot edit the schema box outline manually).

This object-oriented approach also yields additional benefits, as the user can manipulate entire schema objects within the editor in the same way as conventional characters (e.g. the user can cut and paste whole schemas simply by placing the cursor within the schema object). Using an object-oriented approach thereby expands the capabilities of *TranZit* beyond a simple text editor, to a sophisticated capture tool for the Z notation, supporting language features to maintain the consistency of the captured specification.

4.1.4 Accommodating the Learning Potential of the User

Another important characteristic of a mature user interface design, is the ability of the user interface to accommodate the *learning potential* of the user. Initially, the program and user interface should try to lead the novice user through the task of completing

operations within the program environment. However, as the user learns how the program works and becomes more adept at controlling it, the program should not constrain the user to working at this primitive level. Thus the program should offer alternative facilities whereby so-called *power-users* can use the program in a more efficient manner.

To support this view of *TranZit* as a learning aid for the Z notation, elements have been incorporated in the user interface to aid in the selection of specialist Z notation characters. Undoubtedly, one of the most daunting problems faced by any newcomer to Z is the need to understand the mathematical principles of the notation. This is not helped by the strange character set that must be learned in order to express these principles. Whilst *TranZit* cannot teach the mathematics explicitly, it can aid the novice user in locating the relevant characters and providing a basic description.

To this end, Z notation characters are accessible from the *SYMBOLS* menu and are arranged in dialogs according to their functional grouping, to allow users to easily locate the character required.

An example of a symbol selection dialog, highlighting the functional grouping for *expression* symbols is shown in Figure 4-4:

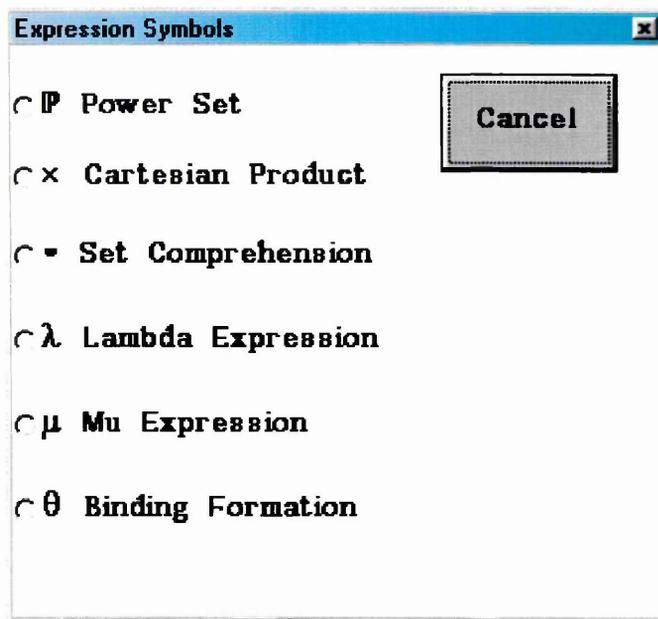


Figure 4-4: TranZit Symbol Selection Dialog

The user may now insert the required symbol at the current cursor location simply by clicking on the symbol, or its description, with the mouse. This menu structure aids the notation learning process by guiding the user from the general functional grouping to the specific symbol required. However, as the user becomes more proficient in the notation, this two-level menu access procedure may become tedious. Therefore, for *power-users*, *TranZit* also makes symbols available as *Windows Accelerator keys*. In this case, all the special characters are available from the standard keyboard using combinations of the ALT and CONTROL keys (for example, the AND character ‘ \wedge ’ is accessed by the *virtual key* ALT ‘A’). Indeed the entire menu system may be navigated in a similar way for those users who have advanced to this level of proficiency. Thus, the power and efficiency of the tool grows with the learning capability of the user.

As a point of interest, in order to increase the efficiency of Windows graphics usage, *TranZit* uses its own *TrueType* font designed specifically for this project, which contains all the standard character set plus additional specialist Z notation characters within the same Windows font object.

4.1.5 Considerations in Designing the GUI for the TranZit Analyser Subsystem

In addition to capturing Z specification within the computer system, *TranZit* includes additional tools to support internal consistency checking of the captured specification. In essence, this involves a syntax analyser and type checker, which together form the TranZit Analyser Subsystem (TAS). However, the mechanisms used to integrate these tools into the TranZit user interface are of great importance.

Similar tools to *TranZit* supporting formal specification, e.g. *Formaliser* (Logica Inc, 1995), take that approach that input to the editor system is *syntax-directed*. In this paradigm, the editor tool uses an internal representation of the Z notation grammar to determine whether the user input is syntactically correct in the context of the current specification state. In this way, the user cannot enter syntactically incorrect constructs, as the tool will automatically prevent this. The argument for syntax-directed editing asserts that the specification is always in a *correct state*, which therefore eliminates the iterative, and sometimes tedious, *edit-compile-correct* cycle associated with most separate syntax analyser/compiler phases.

From a purist viewpoint there is certainly merit to this argument, as formal specification is concerned with ensuring the consistency and improving the quality of specifications. However, recall that the major design principle of the TranZit user interface takes the view that the system should be usable by a novice user. If one is well versed in the Z notation syntax, then a syntax-directed editor is an ideal tool for ensuring that minor mistakes do not slip through unnoticed. However, this approach requires that users possess a fair degree of knowledge of the Z notation syntax before the tool can be used at all.

From the research discussed previously in section 3.2.6 associated with the way we *think* when constructing specifications, it has been noted that many people adopt an almost TDSR approach to constructing a specification, involving an iterative cycle of specification object identification and refinement. Thus, the construction of an abstract specification proceeds very much along the lines of *program design*, as people tend to

find this way of thinking more approachable than beginning from a purely mathematical standpoint. It is interesting to note that even the Z notation itself has evolved to include these more imperative constructs (e.g. the Z construct *if predicate then expression else expression*). It is believed that this is a direct response to the way in which people think during specification construction. It therefore follows that, since *syntactical* correctness is not the primary concern of the majority of users during the early stages of specification construction, it would be an unnecessary constraint of the *TranZit* tool to make syntax-directed input a requirement of the editor system. The *TranZit* system therefore includes separate syntax and type checker phases, which can be invoked at any point in the specification construction process, as the user requires.

The *TranZit* analyser subsystem is also flexible in the way it can be *invoked* to meet the requirements of different users. The associated control dialog is shown in Figure 4-5:

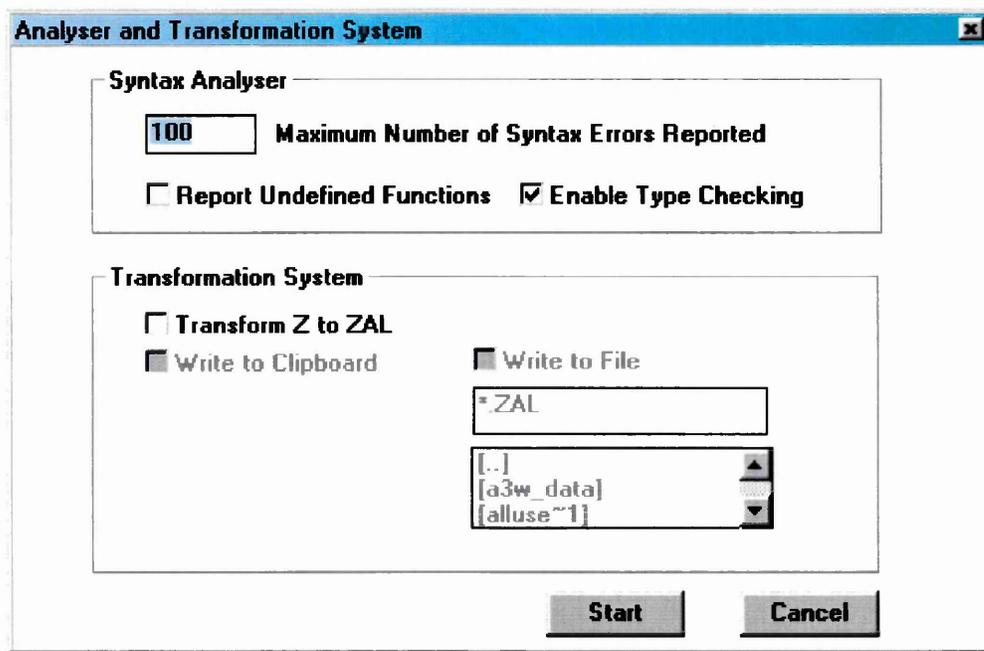


Figure 4-5: TAS Control Dialog

The syntax analyser can be programmed to output a maximum number of errors (meeting the standard user interface requirement that the program output can be adjusted to meet the rate required by the user), and also whether it is to report *undefined* or *implicitly defined* functions (e.g. Z library functions).

In addition, it has been noted from experience with the tool and from teaching exercises that novice users initially find the Z type system more difficult to assimilate than the syntax. There is therefore an option in *TranZit* to independently disable the type system whilst the user concentrates on resolving syntactical problems.

The output of the syntax and type checker is amalgamated into a *TAS Results Window* as shown in Figure 4-6, which is displayed independently of the main editor window.

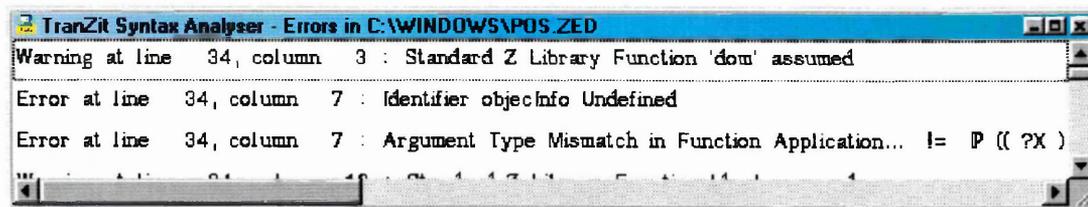


Figure 4-6: TAS Results Window

This gives a detailed description of each problem and its location. Again, to improve efficiency, the results window and editor window are internally coupled such that if the user double-clicks on a line in the results window, the editor window will automatically go to the location in the file where the problem has been identified.

In the case of a type error, the type checker will also generate a *type mismatch summary*, which shows the pure types of the objects it was attempting to resolve at the point when an error was detected. An example is highlighted in Figure 4-7.

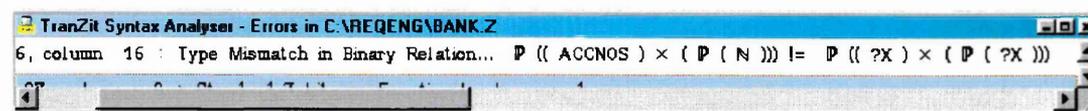


Figure 4-7: TAS Type Mismatch Summary Example

This information can in turn be used to identify the particular abstract object whose type is in error. As shown in the example above, the type checker identifies the generic type it is expecting to the right of the '!=' indication, indicating unbound variables using the

'?X' nomenclature. The bound type string is shown to the left of the '!= ' indicator, showing the mismatch which occurred.

An interesting side-effect of implementing a separate syntax and type checker subsystem, which was not anticipated, was observed when trialling the system with students performing a specification exercise from first principles. It was noted that the users seem to view the syntax and type checkers as a *challenge* to achieving a syntactically correct specification. This appears to give a very clear goal in the specification construction process, and users were in fact heard to use the term "*compiling the specification*", as one would use in the context of general program development. There is little doubt that in their efforts to overcome this challenge, users are forced to address the issue of learning the Z syntax and type system. However, with the approach fostered by the *TranZit* design, this can be taken a step at a time and can be adjusted to grow with the learning capacity of the user.

4.1.6 The User Interface to the TranZit Transformation Engine (TTE)

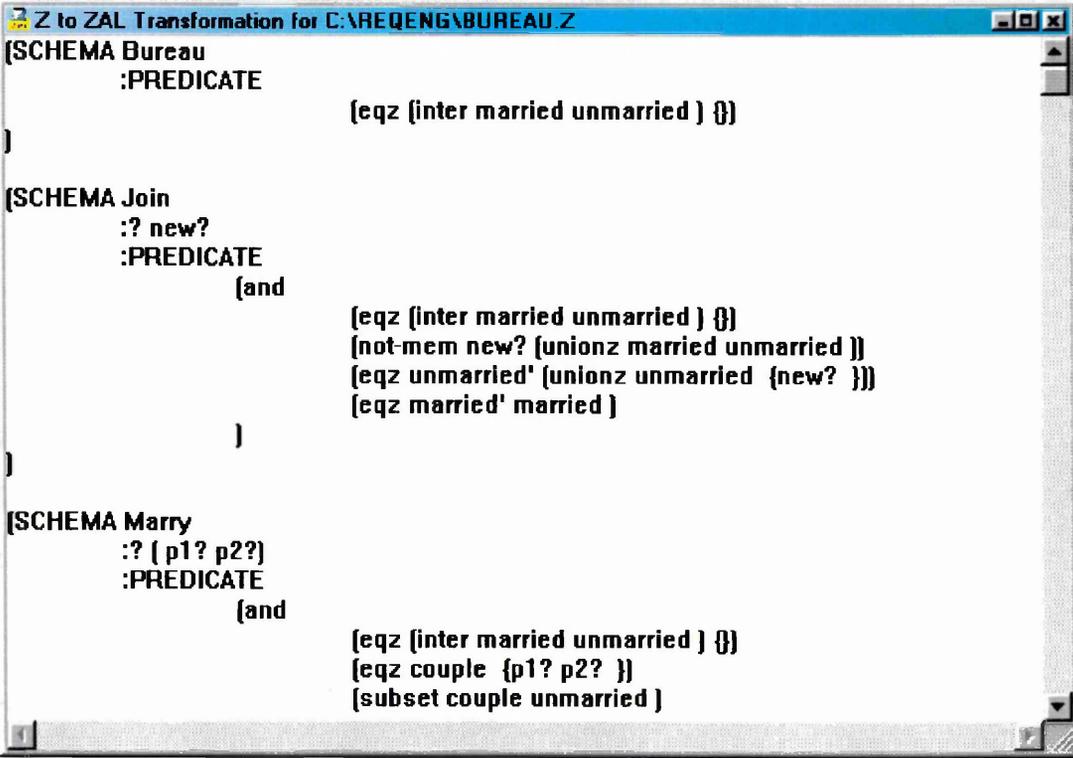
The TranZit Transformation Engine (TTE) is the key tool that integrates *TranZit* with the rest of the REALiZE toolset. This is a sophisticated component which automates the transformation (so far as is practicable) of a captured Z specification into the executable ZAL language (extended LISP) for input to the REALiZE animation tool called ZAL.

A detailed description of the TTE is deferred until Chapter 5: Research and Development of the TranZit Transformation Engine. However, as far as the user interface to the Transformation engine is concerned, this is viewed as an integral part of the process of producing a syntactically and semantically correct specification. Thus the control of the transformation engine is built into the user dialog associated with the analysis subsystem, as shown previously in Figure 4-5.

To ensure the correctness of the transformation and to obviate the need to add additional internal error detection, the transformation engine can only operate on a syntactically and semantically correct Z specification. Thus, the transformation engine will only output

information if the syntax and type checks pass. This output is formatted (the transformation engine output phase incorporates a LISP pretty printer) and can either be stored to a file for input to the ZAL system, or it may be placed on the Windows clipboard for pasting into other suitable applications (e.g. for documentation purposes).

A sample output window showing a transformed specification in the executable ZAL language is shown in Figure 4-8:



```
[SCHEMA Bureau
  :PREDICATE
    [eqz (inter married unmarried ) {}]
]

[SCHEMA Join
  :? new?
  :PREDICATE
    [and
      [eqz (inter married unmarried ) {}]
      [not-mem new? (unionz married unmarried )]
      [eqz unmarried' (unionz unmarried {new? })]
      [eqz married' married ]
    ]
]

[SCHEMA Marry
  :? { p1? p2?}
  :PREDICATE
    [and
      [eqz (inter married unmarried ) {}]
      [eqz couple {p1? p2? } ]
      [subset couple unmarried ]
    ]
]
```

Figure 4-8: Example Transformation Output Window

An important design decision in *TranZit* is that the user cannot edit or manipulate the transformed specification in the ZAL language using the *TranZit* tool. This is because it is paramount in the REALiZE process *that it is the captured Z specification* that is the primary source of information for the system developers. The transformation does not exist to support exploratory prototyping directly in the ZAL language. Rather, it exists to support the requirements engineering agreement task within the REALiZE process, by providing an executable representation of the captured specification for the purposes of animation. If the user were allowed to change the ZAL representation directly, this

potentially creates inconsistencies between the animation results and the original Z specification, which cannot be traced. Thus the whole basis of the REALiZE process would be flawed, as it would not be possible to show that the specification is a true reflection of the customer requirements. Since the process of producing a ZAL transformation is automated in so far as is possible, then there is little effort required to produce a new version from a refinement of the original Z specification, and hence no need to modify the ZAL representation. Similarly (even were it possible), there is no tool to convert a ZAL representation back into the corresponding Z notation, as some semantic information is lost in the conversion to an executable representation which cannot be adequately regenerated. Thus the tool enforces the rule that the transformation process is *one-way* (Z to ZAL) and that no manipulation of the ZAL representation is allowed other than by corresponding changes to the original Z specification.

4.1.7 Evaluating the TranZit User Interface

The research and development of the TranZit user interface has been evaluated by exposure to students and staff at Sheffield Hallam University (SHU) with a variety of experiences in constructing Z specifications. The analysis of these results has been formalised by the production of a questionnaire, which canvasses user opinion concerning the success of the various design decisions made. A discussion of these results is presented in section 6.2.

4.2 Research and Development of the TranZit Syntax Analyser

The next major component in the development of *TranZit* involves the addition of a syntax analyser for the Z notation. The addition of internal consistency checking to the system is a vital component in the process of generating a procedural representation. This is because a procedural representation cannot be developed until the original Z specification is both syntactically (and semantically) correct. In addition, the syntax checking process generates symbol table information, which is required by the transformation process as discussed later. Together with the type checker discussed in section 4.3, the syntax analyser forms a component of the *TranZit Analyser Subsystem (TAS)*.

Syntax analysis is concerned with the *structure* of a language, and not the meaning (semantics). In this case, the essence of the syntax analyser is the ability to *recognise sentences* in the Z notation. That is, the syntax analyser has an understanding of the *grammar* of the Z notation. However, before it is possible to discuss development issues, it is necessary to introduce the concepts and basic definitions (based on Gries, 1971) which will be used in the description of the syntax analyser.

4.2.1 The Parsing Problem

In the first instance, we are concerned with defining, designing and implementing a program that is capable of recognising or *parsing* sentences in the Z notation.

Consider the English sentence "The big dog ate the biscuit". Knowledge of English tells us that this is a sentence of the language. English *grammar* identifies that sentences in English consist of subjects, predicates, nouns and verbs amongst others, which can be put together in a variety of ways in order to construct valid English sentences. In the case of the sentence above it is possible to show the derivation diagrammatically, using the basic rules of English grammar as shown in Figure 4-9:

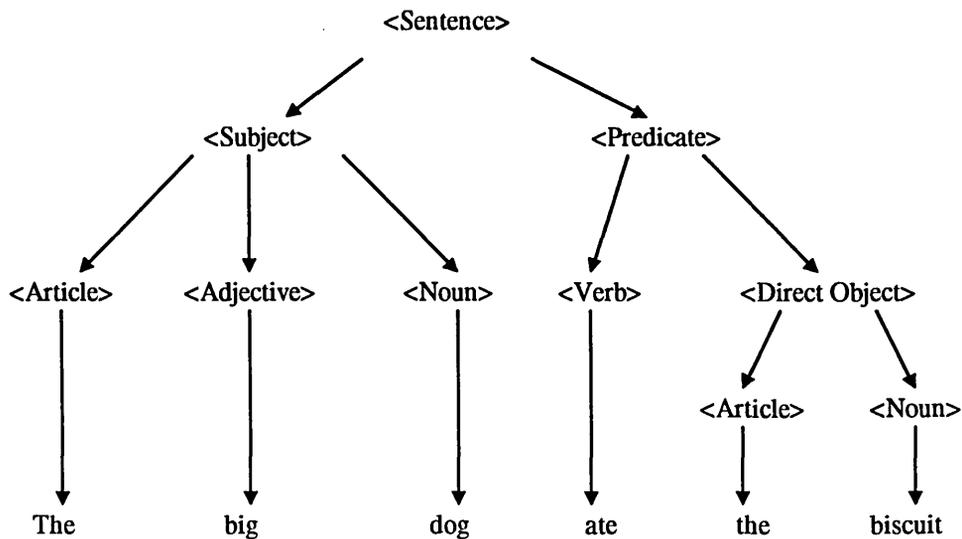


Figure 4-9: Example Syntax Tree

A diagram like the one shown in Figure 4-9 is called a *syntax tree* and describes the syntax or structure of an English sentence by breaking it into its constituent parts. That is *<sentence>* is composed of *<subject>* followed by *<predicate>*, *<subject>* is composed of *<article>* followed by *<adjective>* followed by *<noun>*, and so on.

In order to describe the structure of the language, new symbols or syntactic entities are introduced, such as *<sentence>*, as nodes of the syntax tree. These symbols are enclosed in angular brackets to distinguish them from the basic or *terminal* words of the language. Any node in the tree that has more branches emanating from it is called a *non-terminal symbol*. The nodes forming the leaves of the tree are *terminal symbols*, which are actual words in the language.

To *mechanically* decompose sentences by computer algorithm, it is necessary to define formal and precise rules governing the general structure of the language rather than particular sentences. Such a general language description is called a *grammar*.

It is important to differentiate between the syntax (or structure) of the language and its semantics (or meaning). For example, we could generate a different sentence from the syntax tree shown previously which is syntactically correct, but which is nonsense in English, e.g. "the big biscuit ate the dog". This is because the diagram shown in Figure 4-9 conveys no meaning.

However, diagrams like the one in Figure 4-9 are cumbersome to handle and hence various *meta-languages* have been proposed in order to express grammars in a more succinct manner. The particular notation used in the definition of the Z notation grammar is that due to John Backus (1959).

This particular notation is called BNF, which is an abbreviation for Backus-Normal or Backus-Naur form. In BNF notation, "may be composed of" is abbreviated by the symbol " ::= ". Hence, the information in Figure 4-9 can be represented in Backus-Naur form as shown in Figure 4-10:

```

<sentence>      ::= <subject> <predicate>
<subject>      ::= <article> <adjective> <noun>
<predicate>    ::= <verb> <direct object>
<direct object> ::= <article> <noun>
<article>      ::= the
<adjective>    ::= big
<verb>         ::= ate
<noun>         ::= dog | biscuit

```

Figure 4-10: Example BNF Notation

Once such a set of grammar rules exist, they can be used to derive or *produce* any sentence in the language. For this reason, the rules are often called *productions*. Clearly, some symbol is required to start the production, and in this case the start symbol is <sentence>. The production system operates by identifying a rule with <sentence> to the left of "::=" and proceeds by replacing this symbol with what is on the right. This expansion is shown in Figure 4-11:

```

<sentence>      => <subject> <predicate>
                => <article> <adjective> <noun> <predicate>
                => the <adjective> <noun> <predicate>
                => the big <noun> <predicate>
                => the big dog <verb> <direct object>
                => the big dog ate <article> <noun>
                => the big dog ate the biscuit

```

Figure 4-11: Expansion by the Production System

It is important to note that in the derivation of the sentence the left-most rule is replaced first. This will be important in describing the implementation of the *TranZit* parser.

At this point, it is noted that using this notation, non-terminal and terminal symbols are easily distinguishable without the need for the <..> notation, since a symbol is non-terminal if it appears to the left of a "::=" symbol. Symbols not appearing to the left of "::=" in some production are therefore terminal symbols in the grammar, as they have no production rule to re-write them.

4.2.2 Definitions for Languages and Grammars

From the preceding introductory discussion, definitions can now be made as follows:

Informally a *language* is a subset of the set of all sequences of “words” or *symbols* taken from some basic vocabulary. Note at this point that no meaning is attached to these sequences. An *alphabet* is defined as being a non-empty, finite set of symbols, and a finite sequence of symbols from the alphabet is called a *string*, including the empty string ϵ . Powers of an alphabet A can also be defined as in:

$$A^0 = \{\epsilon\}, A^1 = A, A^n = AA^{(n-1)} \text{ for } n > 0$$

- *Definition:* The closure A^* and positive closure A^+ of set A , are defined as:

$$A^+ = A^1 \cup A^2 \cup \dots \cup A^n \cup A^{(n+1)}$$

$$A^* = A^0 \cup A^+$$

Thus if $A = \{a,b\}$, A^* includes the strings ϵ , a , b , aab , $aaabbbb$, $bbbb$..

It is now possible to define some *rules* or *productions* that organise the symbols of the language.

- *Definition:* A *production* is an ordered pair (U,x) , written $U ::= x$, where U is a symbol and x is a non-empty finite string of symbols. U is the left part, and x is the right part of the production.

Hence, the definition of a grammar follows as:

- *Definition:* A *grammar* $G[Z]$, is a finite, non-empty set of productions. The *distinguished symbol* Z is a symbol that must appear as the left part of at least one production. The set of symbols used in all the left and right parts form the *vocabulary* v . Where it is obvious from the context or the distinguished symbols Z is unimportant, G may be written instead of $G[Z]$.

- *Definition:* Given a grammar G , those symbols appearing as a left part of a rule are called *non-terminals or syntactic entities*. The set of non-terminals is termed VN . The remaining symbols not in the set VN are called *terminal symbols*. These form the set VT such that $v = VN \cup VT$.

It is now possible to define the *language* that corresponds to some grammar. That is, it is necessary to define the sentences that belong to the language. This is achieved by defining three new symbols, \Rightarrow , \Rightarrow^* and \Rightarrow^+ . Informally, $\alpha \Rightarrow \beta$ if we can derive β from α by replacing a non-terminal in α by the right hand side of some corresponding production.

- *Definition:* Given a grammar G , the string α *directly produces* the string β , written $\alpha \Rightarrow \beta$, if $\alpha = x U y$ and $\beta = x u y$, for some strings x and y , where $U ::= u$ is a production of G . Alternatively β is a *direct derivation* of α , or β *directly reduces to* α .
- *Definition:* α *produces* β , or β *reduces to* α , written $\alpha \Rightarrow^+ \beta$, if there exists a sequence of direct derivations:

$$\alpha = \chi_0 \Rightarrow \chi_1 \Rightarrow \chi_2 \Rightarrow \dots \dots \Rightarrow \chi_n = \beta, \text{ for } n > 0$$

The sequence is termed a *derivative of length n*. The string β is said to be a *word* for α .

- *Definition:* $\alpha \Rightarrow^* \beta$, if $\alpha \Rightarrow^+ \beta$ or $\alpha = \beta$.

Informally, a language is simply a subset of the set of all terminal strings VT . The structure of a *sentence* in the language is given by the *grammar*. It is important to note that several different grammars may generate the same language. Hence:

- *Definition:* If $G[Z]$ is a grammar, a string x is called a *sentential form* if x is derivable from the distinguished symbol Z , i.e. $Z \Rightarrow^* x$. A *sentence* is a sentential form consisting of only terminal symbols. Hence the *language* $L(G[Z])$ is the set of sentences:

$$L(G[Z]) = \{x \mid Z \Rightarrow^* x \wedge x \in VT^+\}$$

4.2.3 Language Classes

Formal language theory developed mainly out of the work of Chomsky (1956) who performed much of the early mathematical analysis that led to the understanding of modern computer languages. Chomsky was not directly concerned with the problems of analysing the syntax of a programming language, however the mathematical results generated address a number of issues with the development of a program for recognising grammars. In summary, Chomsky defined different classes of language in which certain properties are evident. The languages are defined in terms of *grammars* that *generate* only languages in that class, and *automata* (or machines) which *recognise* only languages of that class.

Chomsky defines four basis classes of language in terms of grammars, which are 4-tuples (v, τ, ρ, Z) , where v is the alphabet, τ is an alphabet of terminal symbols in v , ρ is a finite set of production rules, and Z is a distinguished symbol (i.e. $Z \in v - \tau$). The difference in the four types of grammars is in the *form* of the production rules ρ .

- *Definition:* A grammar G is *type 0 or phrase-structured* if the rules of ρ are of the form:

$$x ::= y \quad \text{with } x \text{ in } v^+ \text{ and } y \text{ in } v^*$$

That is x can also be a sequence of symbols and the right part y can be empty. In general grammars of this type are of little practical use.

- *Definition:* A grammar G is *type 1 or context-sensitive* if the rules of ρ are of a more restricted form:

$$x U y ::= x u y \quad \text{with } U \text{ in } v - \tau, x \text{ and } y \text{ in } v^*, \text{ and } u \text{ in } v^+$$

The term context-sensitive refers to the fact that U can only be re-written as u in the context $x .. y$.

- *Definition:* A further restriction defines that a grammar G is *type 2 or context-free* if the rules of ρ are of the form:

$$U ::= u \quad \text{with } U \text{ in } \nu - \tau, \text{ and } u \text{ in } \nu^*$$

The term *context-free* refers to the fact that U can be re-written as u regardless of the context in which it appears. Put another way, any U in a sentential form can be expanded using a production of the form $U ::= u$, regardless of what strings surround U in the sentential form.

- *Definition:* A final restriction defines that a grammar G is *type 3 or regular* if the rules of ρ are of the form:

$$U ::= N \text{ or } U ::= WN \quad \text{with } N \text{ in } \tau, \text{ and } U \text{ and } W \text{ in } \nu - \tau$$

Regular grammars play an important role in language and automata theory since languages derived from them can be recognised very efficiently. Unfortunately, regular languages are quite limited and are incapable of describing quite simple programming constructs. For this reason their use is generally associated with recognising basic symbols or *tokens* in a program, forming the basis of a *scanner* or *lexical analyser*.

The four classes of grammars defined are increasingly restrictive; that is there are phrase-structured languages which are not context-sensitive, context-sensitive languages which are not context-free, and so on. The ability to define a language in terms of one of the classes defined by Chomsky allows mathematical analysis of the associated grammar to determine properties of the language. Such analysis then allows the definition of corresponding *automata*, which is an important step in designing efficient and practical parsers for a language.

4.2.4 Grammars and Automata

As already shown, the syntax of the majority of programming languages and notations can be defined using the BNF notation. Indeed, the Z notation itself has been formally

specified in such a way by Spivey (1992) and in the Z Base Standard (Brien and Nicholls, 1992).

It is therefore possible to apply the theory of languages and grammars to the BNF form of the Z notation, in order to determine the properties of an automata to recognise sentences of the Z notation.

It is noted that BNF effectively corresponds to limiting the left-hand side α of each production $\alpha \Rightarrow \beta$ in a *type 0* grammar, to be a single non-terminal symbol. From the previous definition, such a *type 0* grammar with this restriction is a *type 2 or context-free* grammar. Hence the representation of the Z syntax in Spivey (1992) is given in terms of a context-free grammar (CFG).

It can be shown (Rayward-Smith, 1995), that any regular grammar $G = (v, \tau, \rho, Z)$ can be represented as a directed graph with arcs and nodes, such that each node is labelled with an element of v . If there exists a production $\alpha \Rightarrow a\beta$ in ρ , then the node labelled α is connected to the node labelled β with an arc labelled a . Such a graph augmented with an additional start and finish node, describes a *Finite State Automata* (FSM). If there are any nodes with more than one arc leaving it with the same label, then the finite state automata is said to be *non-deterministic*.

- *Definition: A non-deterministic finite state automaton (NFSA) is a 5-tuple, $M = (K, T, t, k_I, F)$ such that:*
 - K is a finite set of states.
 - T is a finite input alphabet.
 - t is a total function $K \times T \rightarrow 2^K$ called the *transition function*.
 - $k_I \in K$ is a designated start state.
 - $F \subseteq K$ is a set of final states.

A *non-deterministic Pushdown Automata (NPDA)*, can be informally described as an NFSA with a stack, the top element of which can influence the transition function. Hence in an NPDA the set of next possible states depends upon the current state, the input

symbol and the symbol which is popped off the top of the stack. When changing to a new state an NPDA may also push any finite number of symbols onto the stack.

- *Definition: A non-deterministic PushDown automaton (NPDA) is a 7-tuple, $M = (K, T, V, p, k_I, A_I, F)$ such that:*
 - K is a finite set of states.
 - T is a finite input alphabet.
 - V is a finite set of stack symbols.
 - p is a total function $K \times V \times T \rightarrow 2^{K \times V^*}$ called the *pushdown function*.
 - $k_I \in K$ is a designated start state.
 - $A_I \in V$ is a designated start symbol on the stack
 - $F \subseteq K$ is a set of final states.

It can be shown (Rayward-smith, 1995), that every regular language $L \subseteq T^*$ can be accepted by some NFSA, $M = (K, T, t, k_I, F)$. From this NFSA we can construct a NPDA, $M^1 = (K, T, \{\perp\}, p, k_I, \perp, F)$, (where \perp is a special bottom of stack marker), such that $T(M^1) = L$. In this case M^1 simulates the action of M by ignoring the contents of the stack.

To ensure this condition, p is defined by:

$$p(k, \perp, a) \text{ contains } (k', \perp) \text{ iff } k' \in t(k, a) \text{ for some input symbol } a.$$

Thus the stack remains at \perp throughout the moves made by M^1 . So,

$$\begin{aligned} x \in T(M^1) & \text{ iff } t_M((k_I, \perp), x) \cap (F \times \{\perp\}) \neq \{\} \\ & \text{ iff } t(k_I, x) \cap F \neq \{\} \\ & \text{ iff } x \in T(M) \\ & \text{ iff } x \in L \end{aligned}$$

This argument shows that every regular language can be accepted by some NPDA. By example, we know that an NPDA can be used to construct a language which is not regular, e.g. $T(M^1) = \{xx^r \mid x \in \{a, b\}^+\}$ for some input a and b , hence we know that NPDA's will accept a strictly greater class of languages than a FSA.

Hence, this result shows that NPDAs are the *acceptors* for Context-Free Languages (CFLs). That is for every CFL, L , such as the Z notation, there exists an NPDA that accepts that language. This is an important result, which will be used in determining the parsing technique to be adopted in the TranZit syntax analyser.

4.2.5 Grammars and Ambiguity

In the example discussed in section 4.2.1, the syntax tree and production rules showed that it was possible to produce two sentences in the language, namely:

- 1) “*the big dog ate the biscuit*”
- 2) “*the big biscuit ate the dog*”

Thus this grammar produces two unique sentences and there is no way within the specified rules in which to derive any other sentence. However, consider the following grammar:

```

<sentence>      ::= <subject> <predicate>
<sentence>      ::= <predicate> <direct object>
<subject>       ::= <noun>
<direct object> ::= <noun>
<predicate>     ::= <verb>
<noun>          ::= time | flies
<verb>         ::= time | flies

```

The problem here is that the sentence “*time flies*” can be generated in two different ways:

```

<sentence>      => <subject> <predicate>
                => <noun> <predicate>
                => <noun> <verb>
                => time flies

```

or alternatively,

```

<sentence>      => <predicate> <direct object>
                => <verb> <direct object>
                => <verb> <noun>
                => time flies

```

Each way, the sentence makes sense in English; "*time flies by very quickly*" or "*go and find out how to time flies*". The problem is that, out of context, it is not possible to tell what the sentence part "*time flies*" means. That is whether 'time' is being used as the verb 'to time' and 'flies' is being used as the plural of winged insect, or 'time' is being used as a noun and 'flies' as the direct object of the verb 'to fly'. Effectively, it is not possible to tell what this part of the sentence means *unambiguously*. Clearly, this would be a major problem for a computer language if a particular statement meant different things depending upon the statements surrounding it.

Hence, if a compiler is to be able to translate all valid source programs in a language, the grammar of the language must be unambiguously defined. Note it is the *grammar* that is ambiguous, not the language.

- *Definition:* If for all $x \in L(G)$, any derivation of x yields the same derivation tree, the CFG G , is *unambiguous*. If however, two or more distinct derivation trees exist for some $x \in L(G)$, G is *ambiguous*.

In some cases, it is possible to rewrite the grammar for a language in such a way that it can produce the same set of sentences in the language, but the grammar is no longer ambiguous. Conversely, there are languages which are inherently ambiguous, that is there is no unambiguous grammar for the language. Unfortunately, on analysis the Z notation as defined by Spivey (1992) turns out to be one of these languages, as explained below.

As Spivey himself points out, the syntax for *set expressions* in the Z notation, as shown below, is ambiguous.

$$\begin{aligned} \text{Set-Expression} & ::= \{ / \text{Expression}, \dots, \text{Expression} / \} \\ & | \{ \text{Schema-Text} / \bullet \text{Expression} / \} \end{aligned}$$

The problem is that if S is a schema, then the expression $\{S\}$ may be either a (singleton) set display, or a set comprehension equivalent to $\{S \bullet \theta S\}$. Spivey makes the point that the expression $\{S\}$ should always be interpreted as a set comprehension and a set display should be written $\{(S)\}$. However, this arbitrary *convention* is not enforced by the grammar definition itself. However, understanding that this problem exists means it is possible to make allowances for it in the parsing algorithm defined for *TranZit*.

4.2.6 Developing a Parser for the Z Notation Syntax

In the preceding discussion, grammars have been used to derive sentences of the language represented by the grammar. However, it is also possible to take a sentence and see if that sentence fits into the grammar of some language. This is called a *parse*. The parse of a sentential form is essentially the construction of the derivation (and possibly the syntax tree) for it, from the grammar of the language.

A parsing program or a *parser* is often called a *recogniser* since it recognises only those sentences that can be derived from the grammar in question. This is of course, the problem at this point; how to recognise specifications written in Z.

Previously in section 4.2.4, it was shown that NPDA's are the acceptors for CFLs. As identified by Rayward-Smith (1995), due to the non-deterministic nature of these machines, a parser for a CFL will always involve some level of *backtracking*. The algorithm itself proceeds deterministically, and at some point is presented with a choice of possible alternatives represented by branches of the syntax tree. This means that it is inherent in the design of the algorithm that the wrong choice will occasionally be made, and backtracking will be required.

There are two ways to approach the task of parsing an arbitrary string $x = a_1a_2a_3.. a_n \in L(G)$. The first approach is to start from the root node and build a syntax tree to work down to the leaf nodes $a_1a_2a_3.. a_n$. This is called a *top-down parse*. Alternatively, it is possible to start with the leaf nodes, and attempt to derive the intermediate syntax tree nodes in order to arrive back at the root node. This is called a *bottom-up parse*.

One of the principle methods used in modern compilers is termed *LL parsing*. This is a top-down approach, although for an LL parsing algorithm to be applied with maximum efficiency to a language, certain constraints are placed on the grammar defining it. In practise these limitations are not too severe and the approach can be applied in a wide variety of compilation problems.

The theory of $LL(k)$ grammars, addresses the problem associated with determining which branch of the syntax tree to select when a choice occurs. This clearly increases the efficiency of the parsing algorithm by eliminating unnecessary backtracking.

- *Definition:* An $LL(k)$ grammar ($k \geq 1$) is one where given any sentential form, $\omega A \gamma$, $\omega \in T^*$, $A \in N$, $\gamma \in (N \cup T)^*$, generated by a left-most derivation, at most a k -symbol look-ahead is required to uniquely determined which of the productions A on the left-hand side should be applied next.

Hence, in an $LL(k)$ grammar the production to apply next relies not only on the input non-terminal symbol A , but also on the next k unmatched input symbols. If this is the case in its own right, the $LL(k)$ grammar is said to be *strong*. However, if the production to apply also relies on the string $\omega \in T^*$ before A in the sentential form, and the string $\gamma \in (N \cup T)^*$ after A , then $LL(k)$ is not strong.

If it is known that the grammar of $L(G)$ is $LL(k)$, then it is possible to write a parser for $L(G)$ using a well-known technique termed *recursive descent*. Although this technique does not use a stack explicitly as for the implementation of a straightforward NPDA, it makes use of the stack implicitly by constructing the parser from a set of procedures that are inherently recursive. It follows that if the Z grammar is $LL(k)$ then it is possible to make use of this technique explicitly.

However, it can also be shown (Rayward-Smith, 1995) that if a grammar $G = (V, \tau, \rho, Z)$ is an $LL(k)$ grammar, then G is unambiguous. Since we know that the CFG representing

the Z notation as presented by Spivey is ambiguous, it follows that this grammar cannot be LL(k).

Computer language designers often strive to represent the syntax of a language in such a way that it is LL(k), or the more important sub-class LL(1). Even if this is not the case, as in the case of Spivey's syntax for the Z notation, it is possible to apply transformations to the grammar rules to bring the grammar closer to LL(k) and thereby make use of standard recursive descent techniques. Even if this cannot be fully achieved, the majority of the parser can then be designed using recursive descent techniques, whilst the remaining syntax rules are dealt with in a more *ad-hoc* manner.

4.2.7 Applying Iteration and Factoring Transformations to the Z Notation Grammar

The basic Z notation grammar is presented in Spivey (1992). On examination of this grammar, several things are immediately obvious. Many of the productions have right-hand sides that contain more than one non-terminal symbol. Moreover the grammar is clearly not LL(k) for any k . To make the grammar more LL(k) in nature requires the elimination of several characteristics of the grammar, so as to reduce the value of k required to determine the next branch of the syntax tree to develop the parse.

For example, examining Spivey's original grammar for the production *Predicate-1*, it is seen that there are three non-terminals on the right-hand side of the production, each representing a possible progression of the syntax tree.

<i>Predicate-1 ::=</i>	<i>Expression</i>	<i>Rel</i>	<i>Expression</i>	<i>Rel</i>	<i>..</i>
				
		<i>Schema-Ref</i>			
		<i>Predicate-1 ...</i>			

Moreover, it is noted that the definition of *Predicate-1* is of the form:-

$$X ::= X | Y | Z$$

Using a left recursive algorithm (i.e. try each possibility in the order given), the first thing which will happen is to call X again, which in turn calls X again, and so on. Thus the parser algorithm would loop indefinitely. However, since the '|' operator is associative, this rule can be re-written as:

$$X ::= Z | Y | X$$

However, this still doesn't solve the problem, as there is no way of detecting loops in Z or Y which might bring the derivation back to X . Moreover, Z and Y might themselves be defined in a similar way, causing regeneration of the original problem. This issue is generally termed the *direct left recursion problem*.

Clearly, the Z notation grammar must be re-written in such a way that these problems do not occur, but which also preserves the original meaning of the grammar.

Two approaches are available (Gries, 1971), which are termed *iteration* and *factoring*. These are mathematical transformations which can be applied to grammar rules, which preserve the meaning of the rule but which help address the problem of direct left recursion and so make the grammar closer to $LL(k)$.

Consider the following rule exhibiting the direct left recursion problem:

$$E ::= E + T | T$$

Thinking about what this actually means leads to the result that the rule for E defines a string consisting either of T on its own, or any number of T 's separated by the '+' symbol. If we introduce some new notation " $\{ x \}$ " meaning "zero or more occurrences of string x ", we can rewrite the rule for E as:-

$$E ::= T \{ + T \}$$

Note that the characters '{' and '}' are simply *meta-symbols* and are not part of the set of terminal symbols used in the grammar.

Hence in the transformed version, direct left recursion has been eliminated from the production without changing the meaning of the rule by the use of *iteration*. This therefore ensures that the parse progresses at this point and does not enter an infinite loop.

A simple example of where this rule can be applied in Spivey's Z notation grammar is the rule for *Expression-2*:

$$\begin{aligned} \textit{Expression-2} ::= & \textit{Expression-2} \textit{Expression-3} \\ & | \textit{Expression-3} \end{aligned}$$

Which can be rewritten using iteration as:

$$\textit{Expression-2} ::= \textit{Expression-3} \{ \textit{Expression-3} \}$$

Such a production is said to be in *Griebach normal form*. More formally, if all the productions in a CFG are of the form:

$$A \rightarrow a\alpha, \quad A \in N, a \in T, \alpha \in (N \cup T)^*$$

then the CFG is said to be in *Griebach normal form*.

If in a CFG there exist *left recursive* productions of the form,

$$A ::= A\alpha, \quad A \in N, \alpha \in (N \cup T)^*$$

it can be shown that if,

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \dots \mid A\alpha_m$$

are all the left recursive productions, with A on the left hand side, and

$$A \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

are the remaining productions with A on the left hand side, then an equivalent grammar can be constructed by introducing a non-terminal A', and replacing all these productions by;

$$\begin{aligned} A' &\rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \dots \mid \alpha_m \mid \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \dots \mid \alpha_m A' \\ A &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \mid \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A' \end{aligned}$$

The second possibility for transformation is called *factoring*. This involves the identification of productions of the form:

$$U ::= x y \mid x w \mid \dots \mid x z$$

Using factoring, the symbol x can be taken out as a factor from the production in much the same way as it would in a mathematical formula. Hence, U is re-written:

$$U ::= x (y \mid w \mid \dots \mid z)$$

where the symbols '(' and ')' are also *meta symbols*.

An example where this can be applied in Spivey's Z notation grammar is part of the *Predicate-1* rule, as shown below:

$$\begin{aligned} \textit{Predicate-1} & ::= X \\ & \mid \textit{Predicate-1} \wedge \textit{Predicate-1} \\ & \mid \textit{Predicate-1} \vee \textit{Predicate-1} \\ & \mid \textit{Predicate-1} \Rightarrow \textit{Predicate-1} \\ & \mid \textit{Predicate-1} \Leftrightarrow \textit{Predicate-1} \end{aligned}$$

Which can be rewritten using *factoring* as:-

$$\begin{aligned}
 \text{Predicate-1} & ::= X \\
 & | \text{Predicate-1 } (\wedge | \vee | \Rightarrow | \Leftrightarrow) \text{ Predicate-1}
 \end{aligned}$$

To eliminate direct left recursion, we can again apply *iteration* to produce an equivalent production:

$$\text{Predicate-1} ::= (X) < (\wedge | \vee | \Rightarrow | \Leftrightarrow) \text{ Predicate-1} >$$

where $\langle \alpha \rangle$ denotes that α is optional.

As can be seen, factoring and iteration also help to reduce the size of the grammar, increasing the efficiency of the associated parser.

Once factoring and iteration transformations have been applied, there can exist at most one direct left recursion right-hand part for a non-terminal in any particular production. If this is the case then the production must be re-written, such that the direct left recursion right-hand part is last in the list of possibilities to try.

For example, consider the following rule for U :

$$U ::= x | y | \dots | z | Uv$$

This rule says that members of the syntactic entity U are x, y or z followed by zero or more v 's. Applying factoring and iteration transformations rewrites the rule for U as:

$$U ::= (x | y | z) \{ v \}$$

Thus, we have again eliminated the left recursion problem, and in this case, the production for U now becomes iterative instead of the previous recursive form, making it much simpler.

Using the iteration and factoring transformation described, Spivey's original grammar has been re-written in $LL(k)$ form, as shown in Appendix I: $LL(k)$ Grammar for the Z Notation. Thus, in designing the parser, these transformations ensure that the production

to apply next relies on the input non-terminal symbol A , and on the next k unmatched input symbols. Indeed the grammar is now almost LL(1), which allows the use of *recursive descent techniques* in the development of a TranZit parser for the Z notation.

4.2.8 Recursive Descent Techniques

A complete description of LL(k) grammars and recursive descent techniques can be found in Backhouse (1979), hence the discussion below is constrained to issues relevant to parsing the Z notation.

If G is an LL(k) grammar, then a parser for G can be written using a technique known as *recursive descent*. In essence this defines a mechanism of implementing a recogniser program for an LL(k) grammar $G = (N, T, P, S)$, such that there is one procedure call pS for every symbol $S \in N \cup T$, where that procedure is designed to recognise any string derivable from S . If $R \in T$, then pR simply checks that the next unmatched input symbol is R .

This type of parser is *goal-oriented*. It predicts that it can execute a sequence of procedures $pX_1; pX_2; \dots pX_m$, which will attempt to recognise the sequence of unmatched symbols $X_1; X_2; \dots X_m$, in the Z specification.

The parser is driven by a *lexical analyser* or *scanner* which converts symbols read from the Z specification into an internal representation termed a *token*. In particular, the TranZit scanner allows for the inclusion of *comments or notes*, enclosed within 'C'-like comment delimiters, to be embedded within the captured specification. This is an extension of the original syntax, but it is considered an important feature for the specifier to be able to improve the readability of the captured specification by supplementing the Z notation with natural language descriptions. The TranZit scanner is a standard state-based implementation supplemented by the additional lexical rules for the Z notation defined by Spivey (1992).

Each sequential procedure pX_i defined in the parser, accomplishes its goal by comparing the next token in the Z specification at the current point in the parse, with the right hand part of the rule for X_i . Other procedures are called to recognise sub-goals for non-terminals in the right hand part of the rule for X_i as necessary.

Even though the transformed Z grammar is suitable for processing by recursive descent techniques, it is still not LL(1). Unfortunately, the actual value of k for this grammar cannot easily be determined. Also, since the Z notation is continually being modified and enhanced, it would be a mistake to implement a parser system which relied on some value of k for the derived grammar as this would be difficult to modify should future enhancements exceed this constraint. The Z notation parser implemented in *TranZit* is therefore designed to deal with an LL(k) grammar, and implements k -lookahead, where the value of k is constrained only by the depth of the stack available to the host machine.

The function of the look-ahead procedure is to deal with the problem of determining the next procedure pS to call, in cases where the syntax tree represented by the grammar at non-terminal symbol R offers a number of possible choices. We assume the existence of some function qS which is identical to pS , but which does not consume symbols in the input stream. The procedure qS essentially stores the state of the derived syntax tree at the current input symbol R , and then proceeds to match input symbols in the same way as pS . The procedure qS will either succeed in matching the sequence of input symbols to the non-terminal S , or it will fail. In either case, on termination, it restores the state of the derived syntax tree and returns the result of this process to its caller function pR . On the basis of this result, pR will then call the function pS , which succeeded in parsing the sequence of unmatched input symbols. The only minor problem with this approach is that qS and its derived sub-procedures must not store semantic information, as this would invalidate the context of the semantic information should the particular parse attempt fail.

This approach assumes that G is unambiguous. Since it is apparent that the Z notation grammar is ambiguous then the approach must be modified slightly. Essentially, it is known that the Z notation grammar is only ambiguous in a very specific context (i.e. Set-expression).

$$\begin{aligned}
 \textit{Expression-3} & ::= && \dots \\
 & | && \textit{Set-expression} \\
 & | && (\textit{Expression-0}) \\
 \\
 \textit{Set-expression} ::= & && \{ < \textit{Expression}, \dots, \textit{Expression} > \} \\
 & | && \{ \textit{Schema-Text} < \textit{set-comprehension} \\
 & && \textit{Expression} > \}
 \end{aligned}$$

The problem arises because of the need to differentiate between parenthesised expressions:

$$(\textit{Expression-0})$$

and a tuple set display of the form:

$$(\textit{Expression}, \dots, \textit{Expression}).$$

To avoid this ambiguity the rule is imposed in the re-written grammar that at least two expressions must appear in a tuple (i.e. there is no way to write a tuple containing less than two components). Similarly, in order to avoid ambiguity with a set comprehension using a schema reference, the list of expressions in a set display must not consist of a single schema reference.

$$\{ \textit{Schema-Ref} < \textit{set comprehension Expression} > \}$$

This must also be the case to avoid ambiguity with a schema reference used as an expression within a set.

$$\{ \textit{Expression}, \dots, \textit{Expression} \}$$

It is therefore possible to write special procedures to deal with these specific contexts, which implement the assumptions identified. Since type checking semantic routines will

also be added, this mitigates any problems associated with erroneous output due to confusion of the recogniser by the ambiguous grammar.

4.2.9 Recovery From Errors

The primary disadvantage associated with recursive descent techniques is that of developing a consistent recovery mechanism, which allows the recogniser to continue to remain synchronised with the input stream when syntax errors are encountered. The problem is that syntax errors can be discovered at any level of the grammar and it is quite likely that at this point the parser is nested some way down within the derived syntax tree.

Most compilers for block-structured programming languages take the view that when an error is found they will simply fail the parse all the way up to the next statement. That is, the rest of the input is ignored until the next BEGIN, END block (in the case of PASCAL) or semi-colon (in the case of C) is found. This prevents additional spurious errors being generated which are simply a consequence of the first error being detected, and the fact that until the parser reaches the leaves of the erroneous syntax tree it has embarked on deriving, it cannot resynchronise with the input stream.

One of the problems in using this technique with the Z notation is that there is no suitable 'statement level' in the Z notation syntax which to fail back to. In general the recogniser cannot resynchronise with the input stream until it returns to the paragraph level, which is much higher in the derived syntax tree than is ideal. Hence it is sometimes the case that the TranZit syntax analyser may generate additional errors, which are dependent on the resolution of some earlier error.

There is no ideal solution to this issue, however in practice this does not seem to cause too much of a problem, providing users appreciate that the errors generated by the TranZit syntax analyser should be resolved in strict sequence.

4.3 Research and Development of the TranZit Type Checker

The techniques described previously have been used to implement the recogniser function of the Z notation parser implemented in *TranZit*. However, to be able to fully *type check* the Z notation specification requires the addition of *Semantic Routines* within the pX_i functions defined by the syntax analyser. Together with the syntax analyser described in section 4.2, the type checker forms the second component of the *TranZit Analyser Sub-system (TAS)*.

Essentially, the type checker embodies functions to store and derive the *type* of objects defined within the specification. In addition functions are provided to support features of the Z notation explicitly such as schema inclusion, scope checking, Δ and Ξ conventions for schemas and a database of signatures for standard Z library functions which may not be explicitly defined. The majority of these functions form interface methods on the *TranZit Schema ObjectBase*, as described in section 4.3.1.

4.3.1 Rationale and Design Criteria for the TranZit Schema ObjectBase

The core data components used to build the semantic routines associated with the TAS reside in an abstract data type termed the *TranZit Schema Objectbase*. The TranZit Schema ObjectBase is an important component in the realisation of the *TranZit* system. Initially, the concept evolved from treating schemas as specification construction objects, which forms the foundation for language intelligence required in the editor subsystem. However, as the project developed, it became clear that this structure could also form the basic database for the analysis and transformation subsystems. The TranZit Schema ObjectBase is shown conceptually in Figure 4-12.

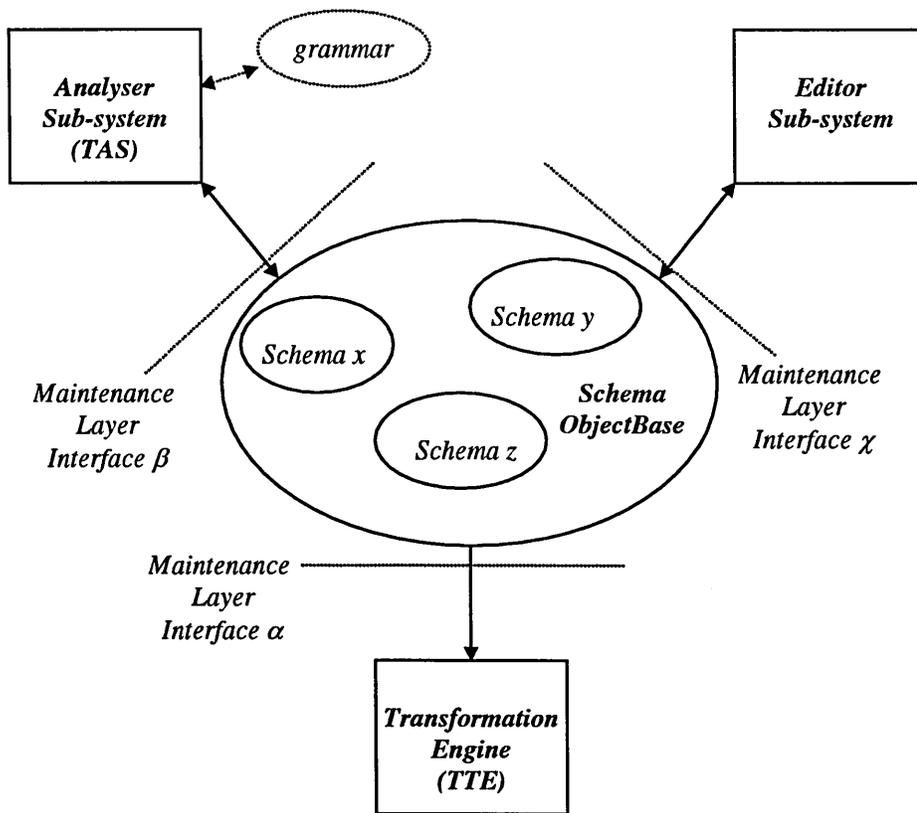


Figure 4-12: TranZit Schema Objectbase Concept

The schema objectbase has evolved during the research of the system, to provide the core interfaces that allow the sub-systems of TranZit to exchange information in an efficient manner. This interface represents the database *Maintenance Layer*, but rather than being homogenous (as in a traditional compiler symbol table), the maintenance layer is structured in an object-oriented fashion to provide different *views* of the objectbase data to different elements of the TranZit system.

For example, to the TAS it presents methods to view the information present as a *block-structured symbol table*, although since there is no equivalent of a “block” in the Z notation syntax it is actually a *schema-structured symbol table*. However, to the editor it presents interface methods that allow schemas to be treated as independent editor objects, and be moved around *en-bloc* within the Z specification. To the transformation engine, it provides symbolic type information used to supplement the productions used

to develop the procedural representation of the captured specification in the ZAL language.

Although the schema objectbase is essentially a database, it is also the core component linking individual sub-systems of *TranZit* together. Using experience gained from traditional compiler design (Holub, 1990), the schema objectbase should therefore have the following characteristics:

- *Speed.* Because the objectbase is referenced every time an identifier or type is referenced, look-up time must be as fast as possible. The entire structure must therefore be memory resident.
- *Ease of Maintenance.* Since the symbol table includes complex data structures, this complexity must be hidden behind a functional or object-oriented interface.
- *Flexibility.* A notation like Z does not limit the complexity of variable declarations, so the design of the symbol table must be able to accommodate any arbitrary type. The symbol table must also be able to grow dynamically as new symbols are added to it.
- *Duplicate entries must be supported.* Because Z allows variables to be introduced locally to a predicate (e.g. by universal or existential quantification), it is possible that these variables have the same name as others declared at higher or lower levels of nesting. The *scoping* rules of the Z notation dictate the set of variables active at any given point in the parse. Hence a distinct symbol table entry is required for each variable, and the objectbase methods must be able to identify the referenced variable from the current scope.
- *Data manipulation.* Methods of the schema objectbase must allow quick deletion or insertion of arbitrary elements or groups of elements within the structure. For example, it must be able to delete all references to local variables associated within a particular predicate, once that predicate has been parsed.

Whilst the concept of the schema objectbase evolved early in the development of *TranZit*, the implementation of the component has been through a number of iterations to improve the speed and efficiency of algorithms it embodies.

4.3.2 Implementing the TranZit Schema ObjectBase

The schema objectbase consists of a number of schema data objects that export methods associated with schema definitions made within the original specification. The editor *creates* objects within the objectbase when a schema is defined. At this point, basic information is stored such as the name, type and generic parameters (if any) of the schema, together with location information which is used by the editor to control presentation aspects of the schema graphic. In the process of checking the schema by the TAS, the objectbase is updated with additional information such as the names and types of variables declared within the schema, which is used by the type checker and ultimately the transformation engine.

The schema objectbase is therefore akin to a *symbol table* in compiler terminology. However, by imposing an object-oriented structure reflecting the view of schemas as *specification construction objects*, this allows a number of Z notation *language-supporting* features to be easily constructed within the editor, analyser and transformation sub-systems. The structure of the schema objectbase is shown in Figure 4-13:

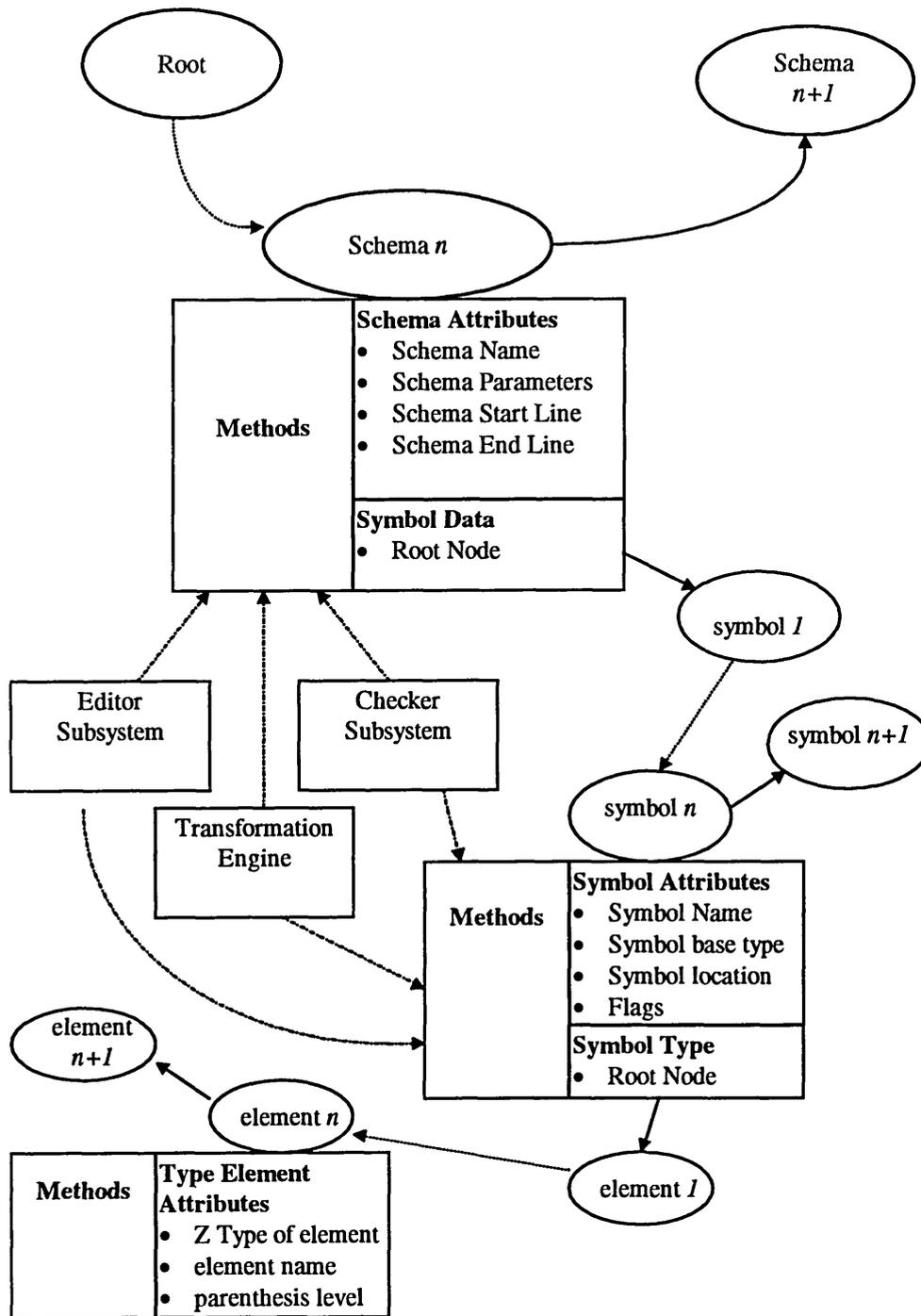


Figure 4-13: TranZit Schema Objectbase Internal Structure

It can be seen that the Schema object base is constructed of three base classes:

- The schema class (*derived types being Schema box, Generic, Axiomatic Definition, Shortform*)

- The symbol class
- The type element class (*derived types being Given Set Type, Power Set Type, Cartesian Product Type, Schema Type, Undefined Type*).

It is easy to extend this mechanism to add another derived class to the objectbase to support global symbols introduced in the Z specification.

Each class exports a number of methods implementing the *Schema Objectbase Maintenance Layer*, which are accessed by the editor, analyser and transformation engine sub-systems to store and obtain information.

Of particular importance is the choice of internal data structure used to store the elements of the objectbase, which are manipulated by the maintenance layer. The wrong choice of data structures will lead to an inefficient implementation, compromising all elements of the TranZit system.

The objectbase itself is constructed from a doubly linked list of object nodes representing individual schemas. This is a relatively simple data structure to implement, and allows for easy insertion and deletion of data. Since the information associated with schemas is fairly constant (other than when new schemas are added or deleted), speed of location of particular schema data objects is not of major importance. Hence the overhead of *scanning* the linked list for a particular schema element does not outweigh the advantage of a simpler data structure.

However, the symbol table associated with each schema object node must be implemented in a more efficient manner. There are a number of techniques that can be used to achieve this. The simplest possible data structure is a stack-based linear array. New symbols are simply pushed onto the stack, and searched for as a list. This mechanism intrinsically supports scoping by the fact that elements at the top of the stack are in scope, and hence are found first. Deletion is also simple, by *poping* as many elements from the stack as required. However, the major disadvantage with this approach is the linear search time required to locate an entry. Also, the maximum size of

the array must be known at compile time, limiting the number of symbols that can be stored in the data structure. Due to the complexity of variable declarations in the Z notation and the limitation of static declaration, this approach was not used.

In many compilers, a *hash table* is used to store the data associated with symbols. In this case the look-up table is implemented as an array indexed by the *key field* of the object stored. The key field contains a *hash value*, which is generated by manipulating the characters of the symbol name in some mathematical way to generate a unique value within the array bounds. To ensure a wide distribution of hash values, some randomisation function is used to ensure that similar names generate very different hash values. Collisions occur when two different symbol names generate the same hash value, and are handled by making each array element the head of a linked list of nodes. Hash tables are very efficient provided that a suitable hashing algorithm can be found to generate a wide variety of hash values for the symbol names allowed within the language. However, hashing was found to be unsuitable for the Z notation, due to the fact that the number of variables defined in the declaration section of a particular schema tends to be relatively small. This wastes quite large amounts of memory within the hash table, which is further compounded by the fact that each schema object node requires its own table. This grouping of symbol names by schema, makes the definition of an efficient hashing algorithm very difficult.

The problems of search time and efficient use of memory can be solved by using a dynamic data structure. The classical data structure for solving this type of problem is a *binary tree*. The average search time in a balanced binary tree is logarithmic rather than linear, in proportion to the number of elements stored. The tree size can grow dynamically as required. However, deletion of an arbitrary node from a binary tree is difficult and time consuming. Fortunately, this is not a problem with the Z notation, as the tree can be constructed entirely within the context of the declarations section of a particular schema, and remains intact whilst the schema itself is in scope. On evaluation, it was decided that a tree structure offered the best compromise between maximising the speed of search and efficient use of memory. However, there are a number of problems with the use of binary trees that need to be addressed.

The first problem with binary trees is the possibility that the data structure will degenerate to a linked list if variables are declared alphabetically and no *balancing* is implemented (Tenenbaum and Augenstein, 1986). In practise, the likelihood of this occurring is small, and so tree balancing was not implemented. The major problem with binary trees is that of *collisions*, in which variables of the same name occur at different scoping levels. This is possible in the Z notation by the *Let* definition, as well as universal and existential quantification, which introduce local variables within the scope of the associated predicate. Since this problem occurs quite frequently in Z, rather than introducing an extension to the schema object node binary tree to include nesting level data, it was decided to adopt a separate mechanism to deal with local declarations. Since the Z notation uniquely identifies when local variables are introduced and destroyed, it is possible for the TAS to inform the schema objectbase when this is the case. In this case, the objectbase maintenance layer stores symbol data in a separate data structure termed the *Local Declarations ObjectArray*, which consists of an array of binary tree roots, indexed by nesting level, as shown in Figure 4-14.

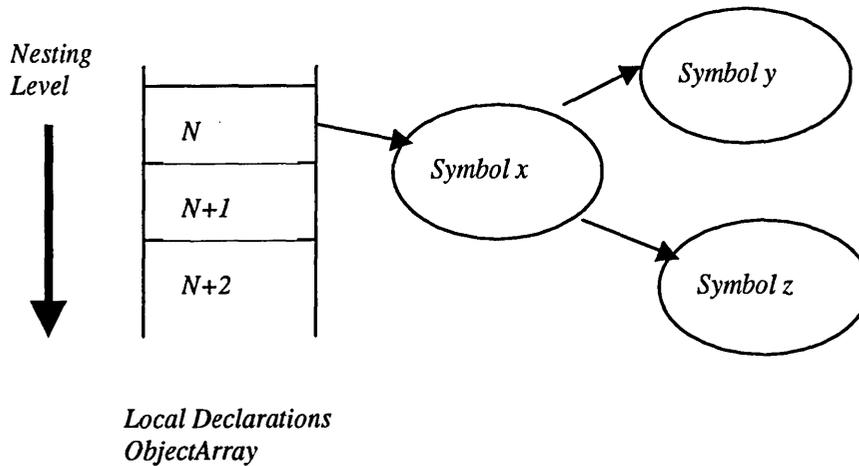


Figure 4-14: Local Declarations ObjectArray Structure

This hybrid approach has been found to be the most efficient in meeting the demands of local variable declarations in the Z notation, by offering a balance of simplicity in handling different nesting levels against speed of search and memory efficiency. The data structure also allows easy destruction of all symbols associated with a particular nesting level by simple deletion of the entire tree associated with that level, rather than having to scan the schema object node binary tree and delete particular elements.

4.3.3 Realising a Type Checker for the Z notation

Every expression that appears in the captured Z specification is associated with a uniquely defined type. It is the strength of the Z type system that allows us to precisely define what is meant by a particular element of the specification, and thereby be unambiguous about what is intended.

The simplest type in Z is a *given set type*, which is used to introduce a set of abstract objects into the specification drawn from some semantic universe appropriate to the problem domain. This also includes the familiar types \mathbb{Z} and \mathbb{R} as members of the set of

given types, as well as the well known given set of natural numbers $\mathbb{N} == \{ n : \mathbb{Z} \mid n \geq 0 \}$.

For more complex types, the Z notation introduces three additional *type constructors*:

- The Cartesian product \times , representing a tuple definition.
- The power set type \mathbb{P} , representing a set of objects.
- The schema type $[\eta_1:\tau_1, \dots, \eta_m:\tau_m]$; where η_i represents a variable name and τ_i represents an associated type. The schema type itself therefore represents a set of mappings of variable name to associated type.

Each of these is conceptually a member of the set *Type* associated with the Z notation. A *Signature* is a function, naturally associated with a schema type, which maps a set of variable names to the set of types in the language.

$$\textit{Signature} \cong \textit{VariableName} \rightarrow \textit{Type}$$

In some languages, the associated type system is simple enough such that the range of this function is a finite set. If this is the case then we say that the associated type system is *constrained*. However, if the set *Type* is infinite, then we say that the associated type system is *unconstrained*. In the latter case the type system allows unlimited complexity in the variable declaration. The Z type system falls into this category, and allows the user to define any complex type from combinations of given set types and the three type constructors.

4.3.4 Representing Types in the Z notation

The fact that the Z type system is unconstrained places certain requirements on the implementation techniques used to store types within the TAS. If the type system were constrained, then it would be possible to represent a particular type by a simple token string within the symbol structure of the schema objectbase. Indeed this is the approach taken by many traditional compilers. However, since this is not the case, then a more sophisticated, dynamic data structure is required.

Since the type of an object can grow almost indefinitely as operators within the specification manipulate it, a linked list of nodes is the most appropriate structure. Referring to the schema objectbase definition, the link list has its root in the symbol node associated with the variable being defined. The type itself is made up of a list of *type element nodes*, each of which indicates one of *given set type*, *power set type*, *Cartesian type*, or *schema type*. For the purposes of computation, we also introduce another type element termed *undefined*, allowing us to represent Z notation generic constants which stand for the as-yet-unknown set of elements which will form the actual definition.

The type is then represented in the computer by traversing the list of nodes associated with the variable name. For example, the type of the *sequence* $\text{seq } X$, which is defined as:

$$\text{seq } X == \{ f : \mathbb{N} \rightarrow X : \text{dom } f = 1.. \#f \}$$

is represented as the sequence of nodes shown in Figure 4-15:

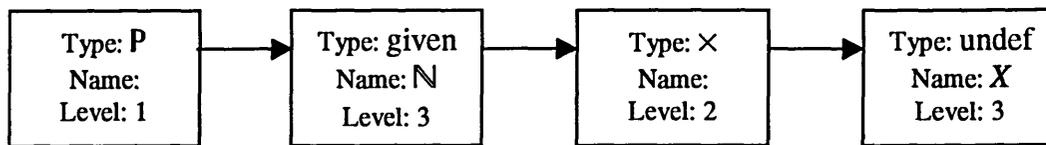


Figure 4-15: Representing Types

There are several important reasons for this choice of representation structure. Firstly, as it stands the actual type of X is undefined, as this is a generic definition, and this fact is explicitly identified within the type representation. The TranZit type checker knows that it can *bind* any actual type to X when a *particular* sequence of this type is declared. At this point the linked list structure makes it easy to remove the node representing the undefined element X , and replace it with the actual bound type in the sequence variable

declaration. Secondly, the *level* information stored within each node is very important as this represents the *associations* between elements of the type list.

Clearly, the type:

$$P (\mathbb{N} \times X)$$

has a very different meaning to:

$$(P \mathbb{N} \times X)$$

The way in which the elements of the type list *associate*, places a completely different context on the composite type. The TranZit type checker understands the rules of association for the type constructor elements, or where provided it builds equivalent structures from parenthesis information. The parenthesis level information is carefully arranged such that elements can be inserted and deleted from the list without needing to revise the context of other *level* information in the list. Thus in the example in Figure 4-15, although it would not appear necessary in this context, the Cartesian product node has a higher level than the nodes for the given set \mathbb{N} and the undefined element X . Hence, when X is bound to an actual variable, the type of this variable can be simply inserted at X without changing the context of the type. Similarly, when building *relations*, it is often the case that we wish to insert a Power Set node P , at the head of the chain. Again this is easily accomplished with a linked list structure by pointer manipulation and associated changes to the level information in each node of the list.

4.3.5 Performing the Type Checking function

Methods associated with the type checker are effectively *semantic actions*, which can be easily inserted into the pX_i functions of the recogniser described in section 4.2.8. In this way, the context of the semantic actions is provided by the grammar itself, allowing the generation of type information from variable declarations and storage of this information in the schema objectbase. In the context of expressions within the specification, information in the schema objectbase is used to provide the types of variables referenced for the purposes of expression type checking.

The TranZit type checker contains two core engines:

- The *expression type builder* (ETB),
- The *type pattern matcher* (TPM).

The ETB is continually constructing the current expression type from information in the schema objectbase, as the parse of the specification is performed. The ETB is actually a set of atomic functions, which are called as semantic actions with the recogniser. The type information being constructed is stored centrally in the type checker, and is manipulated by the ETB. Thus at any stage in the parse the current expression type is always available, together with the previous expression type. This is sufficient to perform all type checking function in Z, due to the binary nature of the relations defined in the language, and also allows a complete type check to be performed in a single pass of the specification. In addition, the ETB is supported by a read-only database of signatures for all operators and common library functions defined in the Z notation, which can be converted on demand into equivalent type element lists.

The TPM actually performs the checking function and is effectively a pattern-matcher, designed to traverse two input expression type lists in sequence, comparing each node for equivalence. An important function of the TPM is *level-balancing*, which ensures that the parenthesis information in each type list is made equivalent (e.g. removal of unnecessary parenthesis) before traversing begins. A second important function within the TPM is the *binding engine*, which attempts to bind types to undefined variables by matching sub-sequences of nodes within one input expression, to undefined nodes in the other. These bindings may be then used further down the pattern matching process to ensure consistency, or made available to other semantic actions to generate actual types from generic elements.

A typical sequence of interactions is shown in Figure 4-16. At some stage in the parse, recogniser function pX_i is called to recognise a declaration that generates some type $Type_i$ for variable Y . This type is built by the ETB and stored within the corresponding element for Y within the schema objectbase. At some later stage, variable Y is used in

some expression associated with recogniser function pX_j . Function pX_j first requests the type of variable Y from the schema objectbase. Assuming, for example, function pX_j is associated with some relational operator Z , it may fetch the current and previous expression type and use the methods of the type generator to build a *relation* R_{ij} from these individual types. It then fetches the stored generic definition of the relation R_z associated with the operator Z , from the signature database, and passes both R_{ij} and R_z to the *Type Pattern Matcher*. The pattern matcher perform level balancing, matching and bindings as appropriate and returns either *TRUE* or *FALSE* to pX_j indicating the result of the type check. If this is successful pX_j builds a new type $Type_j$ dictated by the type rules of the associated expression, which in turn is stored as the new current expression type.

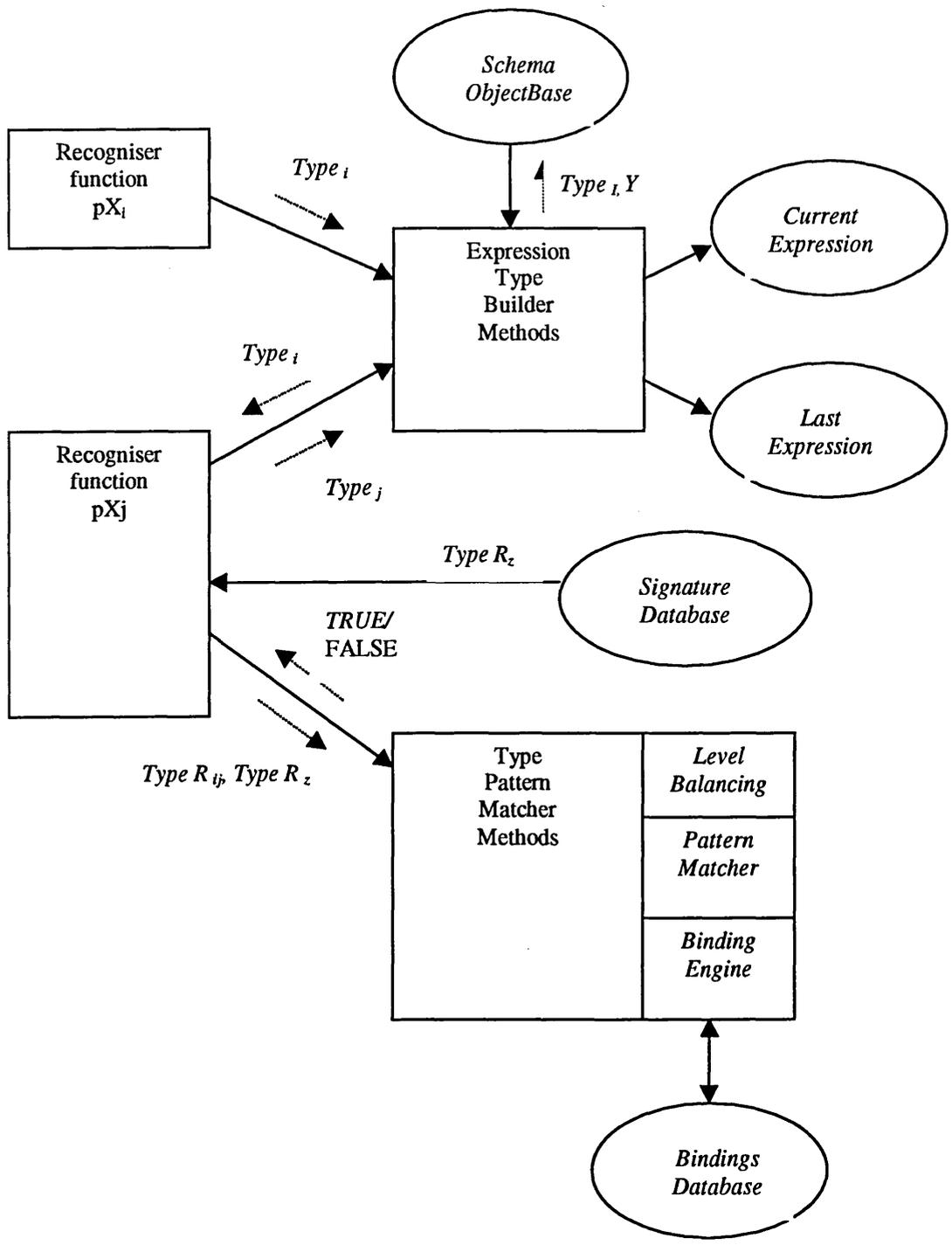


Figure 4-16: Example Interactions within the Type Checker

If type checking fails, pX_j will generate an appropriate error via the standard error reporting mechanism, and invalidate the current expression type. This prevents further type generation within the ETB until the next expression level is reached in the parse,

and hence prevents subsequent spurious type errors caused as a consequence of the original error.

Apart from a few special cases (e.g. build of characteristic tuples), this generic type checking mechanism described can be used by all methods pX_k associated with the recogniser. In most cases this corresponds to pattern matching of relations of the form $P (X \times Y)$, where X and Y are bound to some more complex internal type determined by the declaration of actual variables. The mechanism has proved to be very efficient and powerful since:

- Almost all Z operations are defined in terms of binary relations, and hence can be treated in a standard way.
- The representation of user-defined relations and internal operators/library functions is identical allowing common handling in both cases.
- Expression type propagation is controlled intrinsically from the grammar definition.
- The method inherently allows for the matching and possible binding of generic constants, allowing the type system to be implemented exactly as defined in the relevant texts (Spivey, 1992), and extended at will.

4.3.6 Other Semantic Actions

In addition to the type checking actions embedded within parser functions pX_j , the Z notation demands support for other semantic elements of the notation as follows.

- Variable scoping
- Local declarations (e.g. those made within existential or universally quantified predicates)
- Schema inclusion

In general the rules for variable scoping in Z are quite straightforward. For example, a vertical schema box introduces a global variable for the schema name which must be unique in the specification and which must not be forward referenced. Similarly, an axiomatic definition introduces a global variable of a particular type associated with a set

of predicates governing its behaviour within the specification. Variable declarations or signatures within the declaration section of a schema box are local to that schema definition, and their scope does not extend beyond the schema boundary. However, local variables may also be introduced within a particular predicate using the *Let* definition, existential or universal quantifier, whose scope extends only within the boundaries of the predicate. Variables therefore occur in their binding occurrence in the signatures in which they are declared, and their bound occurrence when they are used within the scope of their binding occurrence.

In general this is all supported by the specialist methods of the schema objectbase. Variables and given sets defined globally, reside in the global section of the database, whilst variables declared within a particular schema are known only to the associated schema object within the objectbase. Similarly, a simple *stack* object controls a stack of data elements in the Local ObjectArray shown in Figure 4-14, which stores information about local variable declarations within schema predicates. This stack object exists only for the life of the current predicate parse and is then destroyed since it is no longer required. A stack is required since it is possible to make nested local declarations in Z with the same variable name.

The complexity of these data structures is hidden behind methods of the schema objectbase. In order to find the correct information associated with a particular variable dictated by its scope, the recogniser functions need only request the schema objectbase start searching at a particular scoping level, either globally, locally within the current schema or locally to the current predicate.

Semantic actions are also required to take account of schema inclusion in the Z notation which allows the variables of one schema to be brought into scope of another schema indirectly. Moreover, a schema containing a schema inclusion, can itself be included in another schema, bringing all variables of the included schemas into scope as well. A further complication allows schema inclusions to be decorated in order to introduce before and after states for the particular variables brought into scope.

At first glance, this problem seems to bring into question our approach to implementing variable scoping, since the variables referenced in the axiomatic part of the schema may be indirectly brought into scope by some schema inclusion in the declarations part. However, the problem is resolved by taking note of the fact that it is not possible to make a forward reference to a schema in Z. For example, a schema cannot be included within another schema until the included schema has itself been declared. Since the act of declaring and parsing a schema builds a corresponding object within the schema objectbase, this ensures that all variable declarations for an included schema exist within the schema object base at the point in the parse where the inclusion occurs.

Due to the object-oriented nature of the schema objectbase, the only real way to deal with schema inclusion is to provide a method which *copies* the contents of the included schema symbol table into the current schema symbol table (taking account of any decoration within the inclusion). The symbol table for this schema object now contains not only its own declarations, but also the declarations of the included schema object. If this schema is itself the subject of a schema inclusion, then all the variables in scope of that schema are automatically brought into scope by the copy process. This is easily implemented in the schema objectbase maintenance layer, by copying the contents of the binary tree associated with the *included* schema, into the binary tree of the *including* schema.

The only negative side to this approach is that there is duplication of memory usage, however this is considered acceptable in order to retain the overall object-oriented approach to symbol manipulation.

4.4 Summary

In this chapter we have explored the research and development of both the TranZit editor subsystem and also the TranZit analyser subsystem (TAS).

The innovative use of traditional compiler design techniques coupled with novel data structures and methods applicable to the Z notation has resulted in a highly compact and

efficient implementation capable of detecting a wide variety of errors within Z specifications.

The analyser subsystem has been exposed to a wide variety of problem domains and many student projects, giving a high degree of confidence in the implementation. Whilst some may argue that this can only be proven by formally specifying the analyser subsystem itself coupled with rigorous testing, the evolutionary and practical approach to this problem has demonstrated the accuracy of the implementation beyond reasonable doubt.

In addition, the exposure of the system to a wide variety of users has produced some interesting insights into the thought processes used in the construction and checking of Z specifications themselves. In particular, the view expressed by many people using *TranZit* of Z being almost a “formally-defined programming language”, stimulated a goal-oriented approach to specification construction in many users, which was not anticipated. This goal-orientated approach of “making the specification compile” seems to have allowed many users to better understand the Z notation by structuring their approach to developing Z specifications around the *TranZit* tool.

In addition, the act of eliminating errors in the specification offers definite learning benefits. When faced with definitive errors produced by *TranZit*, users are forced to address the intricacies of the Z notation in a structured way, in order to get the specification to parse successfully and thereby proceed with the REALiZE animation process.

With these thoughts in mind, the next chapter considers the research and development of the *TranZit* transformation engine, which is capable of automating (so far as is possible), the conversion of a captured Z specification into an executable representation in the ZAL language.

5. Research and Development of the TranZit Transformation Engine

This chapter describes the research and development of an innovative *transformation engine* capable of transforming Z specifications captured in *TranZit* into an executable representation in the ZAL language.

The chapter discusses the approach to transformation adopted by *TranZit* and draws parallels between transformation and code generation in a conventional compiler. Firstly, the evolution of the ZAL language itself is discussed, which provides executable representations for a subset of Z notation operators. This information is then used to derive a CFG for the ZAL language, which in turn defines the target of the transformation process. Having identified the source and target grammars, the discussion proceeds to the transformation process itself. The essential design decisions are outlined, followed by a technical systems analysis of the components of the TranZit transformation engine. Finally, the approach taken by *TranZit* in resolving the transformation of non-computable constructs in the Z notation is described.

5.1 The Rationale for Transformation

The previous chapter described TranZit's approach to processing the captured Z specification for the purposes of checking internal consistency. Although innovative approaches have been identified to achieve this goal, in itself the development of the TranZit Analyser Subsystem (TAS) is not the primary goal of this project.

The essence of the REALiZE process proposed in this thesis, is the ability to validate the captured specification by *demonstrating properties* of the system represented by the specification to the user. The process seeks to support the view proposed by several commentators (McCracken and Jackson 1981, Gladden 1982, Agresti 1986) that the sequence of events in the traditional software lifecycle is unrealistic in assuming that specifications can be frozen early in the development lifecycle. Similarly, the traditional lifecycle fails to recognise the importance of feedback and iteration in validating the

specification, until some software system components have actually been constructed. The all-important *user-view* is therefore often injected too late into the development process, which may engender costly re-design. Therefore techniques are needed to improve the interface between the specification and design phases and in particular provide early feedback on the specification validity from *a user perspective*.

As discussed previously in section 1.2.7, suggested techniques to achieve this ideal incorporate ideas such as *rapid prototyping* and *animation* in order to promote *validation by execution*.

The benefits of these approaches have been identified by Fuchs (1992) and include:

- The availability of executable components much earlier than in the traditional lifecycle, thereby affording earlier (less expensive) detection and correction of problems.
- Requirements that are unclear can be clarified by *interaction* with the specification.
- Execution of the specification supplements inspection and formal reasoning as a means of validation.

The term *prototyping* has a variety of meanings. In some cases it refers to a cut-down version of the final system which has the appearance of the final product, but has limited functionality. In its simplest form it may be a pure mock-up intended to demonstrate how the final system may look, but which contains no actual system components. Such prototypes can often be constructed using the abstraction power of 4GLs, and the speed at which this can be achieved has led to the term *rapid prototyping*. A well documented example of a prototype being used in the context of specification verification is that of Henderson (1986), who propose a mechanism based on a notation known as *me-too*, in which the *me-too* representation is viewed as both a formal specification and an executable prototype.

Whereas a prototype embodies some concept of software development, the term *animation* refers to the ability to ensure adequacy and accuracy of the specification by reflection of some specified behaviour back to the user (Kramer and Keng, 1988).

Animation is therefore an *interactive* concept, which is open to implementation in a number of ways, the approach taken here being akin to that of *executable specifications*. In other words, the essence of our validation process is the *transformation* of the captured specification to an executable representation, which can be exercised in an animation environment.

5.1.1 Constructing an Executable Representation of a Specification

The term *executable specification* is something of a misnomer in many cases, as highlighted later in this chapter, since the majority of formal specification languages are not directly executable. Therefore, the primary consideration to support the animation environment is the selection of a suitable executable *representation* of the original specification.

There have been different approaches taken in attempting to realise executable specifications. The method adopted in this research is to retain the original specification notation (i.e. Z) intact, and map the operational semantics of the notation to a suitable executable representation termed the *Z Animation in LISP* (ZAL) language, which is in turn based on extended LISP.

Other researchers have taken similar approaches using declarative languages such as Prolog (Dick *et al.*, 1990), Miranda (North, 1990), Ada (Moulding and Newton, 1992), Me Too (Henderson, 1986) and Functional languages (Johnson and Sanders, 1989) with varying degrees of success. Executable languages are inherently less expressive than abstract specification languages, since their functions must be computable and their domains must have a finite representation. As highlighted by Breuer and Bowen (1994), any approach must balance declarativeness against efficiency in that whilst the executable representation is considered to be a high-level specification, it must also be able to execute efficiently if it is to be useful as a vehicle for validation.

The essence of the approach taken herein focuses on identifying elements of the Z notation which have executable representations, and modelling these as a library of predicates/functions in the chosen programming language. Whilst this approach provides

a mechanism to address a fair proportion of specification problems, it may still be the case that a naive transformation fails to execute due to an attempt to search an infinite address space, or may fail to find a solution in reasonable time due to inefficiency. Knott and Krause (1992) have suggested ways to address the latter problem using program transformation techniques applied to Prolog.

An alternative approach is to restrict the specification notation to a known operational subset of the original language, which can either be interpreted directly or cross-compiled to a suitable programming language (Doma and Nicholl, 1991). Perhaps one of the main advocates of this approach is Valentine (1995), who has presented several papers on the Z-- language, which is an executable subset of the Z notation. The advantages of this approach are the increase in efficiency of translation and execution time, however this is gained at the expense of loss of abstraction and expression in the restricted specification notation.

It is interesting to map these different approaches by expressibility of the resulting transformation (i.e. a measure of the abstract power of the resulting transformation), and executability (i.e. how easily the transformation may be executed on a computer), as shown in Figure 5-1.

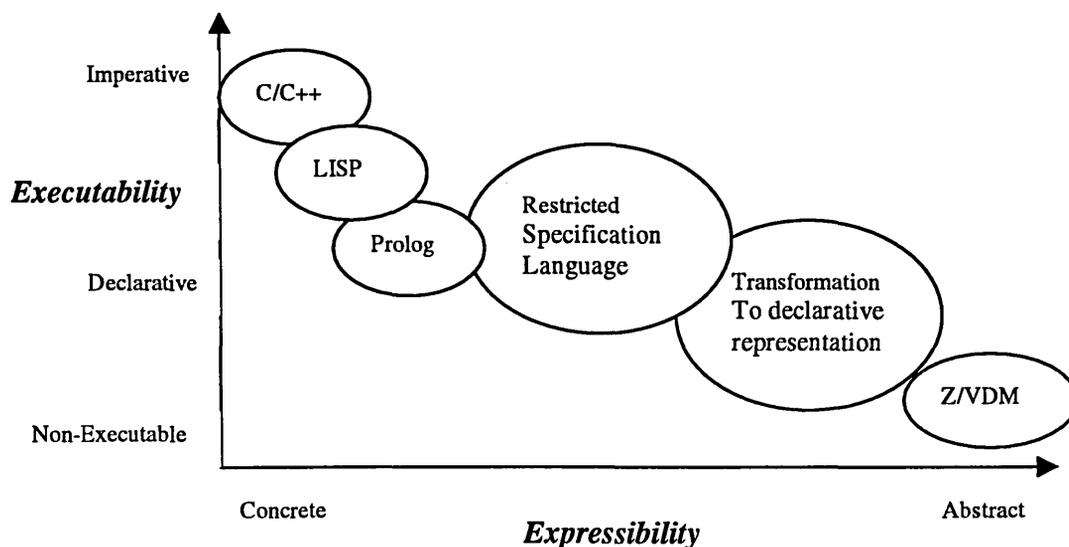


Figure 5-1 : Comparing Executability and Expressibility

It can be seen that in general, as we move to a more concrete specification model, so the possibilities for execution increases at the expense of expressibility.

In order to try to retain expressibility with increasing levels of executability, some hybrid approaches have been attempted (West and Eaglestone 1992, Hasselbring 1994), in which the original specification is re-written or refined into an executable subset of the original language, having the same meaning.

5.1.2 Evolution of the Z Animation Language (ZAL)

Hekmatpour (1988) originally suggested extensions to the LISP Programming language (Wilensky, 1986) which could be used to implement some of the features of functional programming languages such as Miranda (Turner, 1985). The starting point for the evolution of the ZAL language (Siddiqi *et al.*, 1991) was the extension of this idea to consider similar extensions to LISP in order to transform Z specifications into executable representations. From this beginning, the Z Animation in LISP (ZAL) language has evolved to incorporate many operational features of the Z notation, whilst retaining three important design criteria.

- To preserve as closely as possible, correspondence between the original Z notation specification and the ZAL animation language. This ensures that the abstract power of the specification notation is preserved as far as possible, and that the transformation process is not needlessly complicated.
- To ensure that ZAL and Common LISP can be intermixed in an animation in order to capture the existing power of the LISP language, and avoid unnecessary duplication.
- To ensure that the ZAL language is generic, in that it does not impose arbitrary constraints concerning the classes of problem which can be represented. This ensures that the ZAL language can be used to address as wide a variety of specification problems as possible.

It is important to draw a distinction between approaches associated with defining an executable subset of Z (Valentine, 1995) and the development of the ZAL language. In

the former case the emphasis is on *constraining* the use of the Z notation in order that the original Z specification becomes directly executable. However, in the case of the ZAL language, emphasis is placed primarily on *emulating Z operators* in a programming language, without concern for the executable capability of particular Z constructs. As discussed later, it is therefore possible to combine individually executable ZAL operators into a construct that may not then be executable as a whole. However, this *operational* approach is an important aspect of the ZAL language, in that since it is not constrained by considerations of executability, it retains a level of abstraction that is close to the original Z notation. In this system, the task of identifying executable Z constructs is located in the TranZit transformation engine, which in turn creates the relationship between the Z universe and the executable representation.

The ZAL language has been developed over a number of years coupled with the development of the associated ZAL animation environment. This is based on the PC Allegro LISP platform, and involves the research and development of execution mechanisms and interfaces that allow animation scenarios to be explored. As stated previously since this work is the subject of a parallel research programme, the development of the ZAL animation tool itself will not be described herein, other than to appreciate that ZAL and the associated animation environment are based firmly in the procedural domain. For further information in this regard see Morrey *et al.* (1998) and Siddiqi *et al.* (1997, 1998).

However, in order to describe the TranZit Transformation Engine developed for the *TranZit* tool as part of this project, it is necessary to describe the set of Z operators that are modelled in ZAL, as shown in Table 5 below.

Z Notation	ZAL Keyword	Description
\rightarrow	makemap	<i>Total Function</i>
R^{-}	inverse	<i>Relational Inversion</i>
\triangleleft	domres	<i>Domain Restriction</i>
\triangleleft	domsb	<i>Domain Anti-Restriction</i>

\triangleright	ranres	<i>Range Restriction</i>
\triangleright	ransub	<i>Range Anti-Restriction</i>
<i>dom</i>	dom	<i>Domain</i>
<i>ran</i>	ran	<i>Range</i>
\oplus	override	<i>Function Override</i>
(,)	mkt	<i>Tuple</i>
\mathbb{P}	powerset	<i>Power Set</i>
$<$	lessz	<i>Less than</i>
$=$	eqz	<i>Equality</i>
\mathbb{N}	fatn	<i>Set of Natural Numbers</i>
\neg	not	<i>Logical inversion</i>
\vee	orz	<i>Logical OR</i>
\wedge	andz	<i>Logical AND</i>
\exists	exist	<i>Existential Quantifier</i>
\forall	forall	<i>Universal Quantifier</i>
\exists_1	exist-one	<i>Unique Quantifier</i>
\Rightarrow	imply	<i>Implication</i>
<i>disjoint</i>	disjoin	<i>Disjoint</i>
\langle , \rangle	mkq	<i>Sequence</i>
\circ	rel-compose	<i>Relational Composition</i>
$_ _ _$	rel-image	<i>Relational Image</i>
{ , }	mks	<i>Set Display</i>
{D P•E}	mksi	<i>Set Comprehension</i>
#	card	<i>Cardinality</i>
\notin	not-mem	<i>Not Set Member</i>
\cup	union-dis	<i>Distributed Union</i>
\cap	inter	<i>Intersection</i>
\mapsto	make-maplet	<i>Maplet</i>
\cup	unionz	<i>Union</i>

\setminus	setsub	<i>Set Subtraction</i>
\cap	inter-dis	<i>Distributed Intersection</i>
\subset	psubset	<i>Proper Subset</i>
\in	mem	<i>Member</i>
\subseteq	subset	<i>Subset</i>

Table 5: Table of ZAL Operators

Since the ZAL language is an extension of LISP, it is also possible for the TranZit Transformation Engine to make use of standard common LISP operators and constructs in an executable representation. For example, the standard LISP ' \leq ' *less-than-or-equal* operator can be used for the Z notation ' \leq ' operator. Similarly the LISP (*if test-form then-form [else-form]*) construct can be used in the context of the Z notation expression syntax "*if predicate then expression else expression*".

Although ZAL is an extension to LISP, the transformation engine in *TranZit* takes the opposite view and treats LISP as an extension to the ZAL language. The transformation engine will therefore only use appropriate standard LISP operators where there are no corresponding ZAL constructs.

In addition to the standard Z operators, ZAL contains meta-language symbols such as (SCHEMA), which allow these operators to be grouped into executable units equivalent to Z notation schemas. ZAL also allows *inclusion* of such units within other units, thereby modelling schema inclusion. Finally, ZAL is supported by monitoring features such as "SHOW var", which allow the state changes associated with a particular abstract data object to be displayed to the user as execution progresses.

A straightforward example of a small Z specification, and the associated executable representation in ZAL is given in Figure 5-2. The Z schema is given first, below which is the associated executable representation in the ZAL language. The ZAL representation also contains notes in *italics* that have been added for descriptive purposes.

ClassageOf : NAME \rightarrow \mathbb{N} #ageOf \leq 30 $\forall a : \mathbb{N} \mid a \in \text{ran ageOf} \bullet a > 21$

(SCHEMA Class
:PREDICATE
(and

Schema Class in ZAL
start of predicate section
the result is the conjunction of each predicate modelled

```
(\<= (card ageOf )30 )
(forall a (ran ageOf )
  (imply
    (mem a (ran ageOf ))
    (\> a 21 ))))
```

Update **Δ Class**

n? : NAME

a? : \mathbb{N}

a? > 21

(n? \in dom ageOf \vee (n? \notin dom ageOf \wedge #ageOf < 30))ageOf' = ageOf \oplus {n? \mapsto a?}

(SCHEMA Update
:? (n? a?)
:INCLUDE delta_Class
:PREDICATE
(and

Schema Update in ZAL
identify input variables
include a schema definition from elsewhere

```
(\> a? 21 )
(or
  (mem n? (dom ageOf ))
  (and
    (not-mem n? (dom ageOf ))
    (\< (card ageOf )30 )
  )
)
(eqz ageOf' (override ageOf { #(n? a? ) }))))
```

Lookup \exists Class n? : NAME a! : \mathbb{N}
n? \in dom ageOf a! = ageOf(n?)

(SCHEMA Lookup	<i>Schema Lookup in ZAL</i>
:? n?	
:! a!	<i>define output variables</i>
:SHOW a!	<i>Show state changes of this variable to user</i>
:INCLUDE psi_Class	
:PREDICATE	
(and	
(mem n? (dom ageOf))	
(eqz a! (applyz ageOf n?)))	

Figure 5-2: Example Specification with Corresponding ZAL Representation

5.1.3 Development of the ZAL Grammar

From the perspective of the TranZit transformation engine, ZAL and LISP provide a set of procedural operators associated with particular Z operators, which can be used to build an executable transformation. These operators are essentially terminal symbols in some context-free grammar (CFG). In order to perform the transformation, it is required to know how these symbols need to be organised to form an executable representation acceptable to the ZAL animation environment. That is, it is necessary to derive a CFG for the ZAL Language itself.

This is essentially an empirical process, which can be achieved by a series of reifications of the original Z notation grammar as follows:

- 1) Examine the original Z notation grammar to identify syntactic elements that map to operational constructs modelled in the animation environment.

- 2) Analyse the semantics of the identified Z Notation to determine to what extent factors such as declarations need to be modelled in the chosen elements.
- 3) Determine whether corresponding constructs can be generated in ZAL, extended by native LISP where necessary.
- 4) Map ZAL and LISP terminal symbols to the remaining constructs.

The derived CFG for the ZAL language is shown in Appendix II: Context-Free Grammar of the ZAL Language.

There are several facets of this grammar, which are of consideration in the transformation process. Firstly, since ZAL is based on LISP, it is noted that the operators of the grammar are presented in a *polish* style. That is all productions containing operator terminal symbols have those symbols as the first symbol in the production. Since this is not the case in the Z notation grammar, which uses in-fix, pre-fix and post-fix operators, there is clearly a polish conversion function to be considered as part of the transformation process.

Secondly, it is noted that the ZAL grammar includes no declaration syntax, other than to define the input (?) and output(!) variables associated with schema declarations. This is a consequence of the fact that given sets, global and local variables are considered elements of the candidate data set associated with a particular animation scenario. They are therefore provided in the animation environment as needed. However, this does not imply that the transformation process need not be concerned with these elements of the original Z specification. In particular, it is noted that the transformation process must consider the particular type of an identifier in order to produce a correct transformation. Thus, even though *type* information associated with identifiers is not a component of the transformed specification, it is an integral component in the *selection* of the correct production to apply in the transformation process, as discussed later.

Finally, it is noted that not all the operational elements of the Z notation are currently emulated in the ZAL language. Some operations are not yet supported, whilst other elements are implemented by corresponding operations in the native LISP language. The

transformation process must therefore implement a strategy whereby ZAL language elements are selected if available, followed by native LISP expressions where not. There is also the possibility of no corresponding operational transformation in either ZAL or LISP, whereupon the transformation process must highlight this problem to the user.

Having defined the grammar of the ZAL language, it is now possible to discuss the process of *transformation* of a captured Z specification into the ZAL language. In essence, this process is a *mapping* between the grammar of the Z notation and the grammar of the ZAL language. However, this mapping is not a simple *translation*, but a *transformation* in which factors other than simple pattern recognition play an important part. This transformation process is achieved by a novel element in *TranZit* termed the *TranZit Transformation Engine (TTE)*.

5.1.4 Evolution of the TranZit Transformation Engine (TTE)

The TranZit Transformation Engine (TTE) provides the core interface between the *TranZit* system and the ZAL animation environment as shown in Figure 5-3.

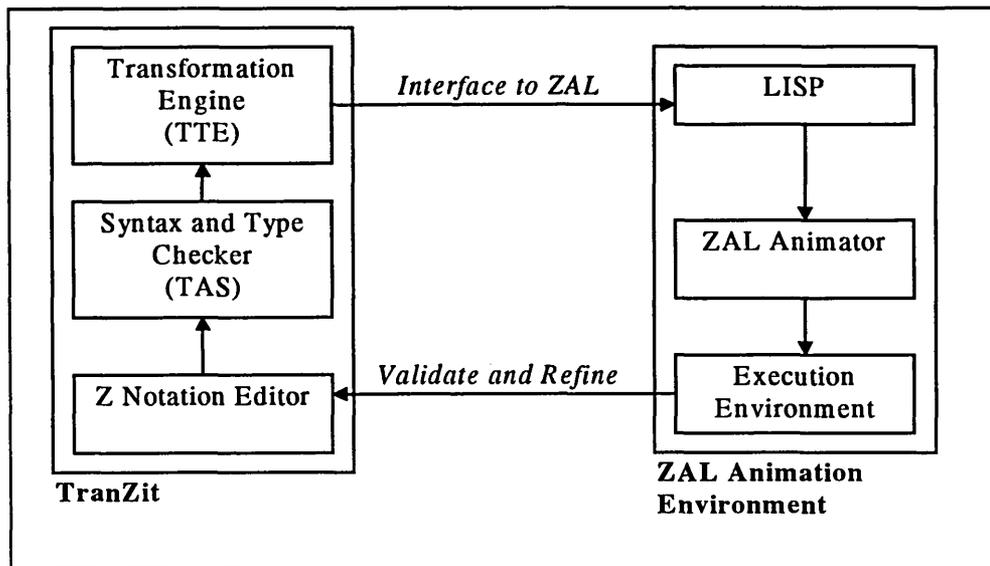


Figure 5-3: The Interface between TranZit and the ZAL Animation Environment

Before *TranZit*, the process of animation was undertaken by transforming Z specifications into the corresponding ZAL language by hand. However, this method provoked a number of problems:

- The process of hand transformation is prone to error, thereby weakening the formalism employed.
- The process can be laborious and time consuming for a non-trivial specification. This is especially important when considering that the REALiZE process inherently incorporates the notation of specification refinement by iteration of successive animations.
- A specifier wishing to use the animation process needs to learn not only the Z notation, but the ZAL language as well. Since the ZAL representation exists purely to support the animation process, this is inefficient use of skills.

The TTE was therefore developed in order to *automate*, in so far as is possible, the transformation of the captured Z specification into the ZAL language. It is important to recognise at this stage that an automated transformation of all possible Z notation constructs is a *non-computable* problem, since in some cases it is necessary to *reason* about the meaning of a Z construct in order to deduce a transformation which will terminate when executed. Turing's *halting problem* tells us that we cannot write a program to achieve this (Wulf *et al.*, 1981). However, since this problem was recognised at the outset of the TTE development, additional mechanisms have been implemented to supplement the automated approach.

5.1.5 Requirements for the TranZit Transformation Engine

The requirements for the TranZit Transformation Engine (TTE) are as follows:

- The user should not need to make any changes to the originally captured Z specification in order to use the TTE. That is, the user is free to write the Z specification using any legal constructs in the Z notation, without concern for executability.

- The TTE will transform a Z specification to a representation that can be executed directly in the ZAL animation environment, with as little user intervention as possible.
- The TTE will provide a mechanism for detecting and dealing with some non-computable clauses in the Z notation.
- The TTE will incorporate features for generating animation-specific constructs within the executable representation, suitable for supporting introspection of the behaviour of the executable representation (e.g. the ZAL **:SHOW** command).

It is important to note that these requirements enforce the view that, in writing the specification, the specifier need not be concerned with writing in such a way as to afford possible executability. Determining executability is the job of the transformation engine, not the specifier.

Having established the requirements of the TTE, certain fundamental decisions needed to be made concerning the mechanisms required to achieve them: Of most importance were the need (or not) for an intermediate language format, accessibility of the executable representation and guaranteeing correctness in the transformation process. These issues are discussed below:

5.1.6 Intermediate Languages

One of the earliest research debates concerning the implementation of the TTE, concerned the necessity to employ an intermediate language between the original Z specification and the executable representation in the ZAL language. Traditional compilers often use an intermediate, idealised language, which is optimised for some unrealised *virtual machine*. There are several advantages to this approach:

- The output of the compiler can be tested in isolation from the target by simulating this virtual machine in software.
- The intermediate language can be designed to alleviate target language problems and allow the compiler designer to focus on generic solutions to problems such as optimisation.

- There is an abstraction between the intermediate language and the target language, allowing different target languages to be generated from the same intermediate language. A separate phase of the compiler, normally termed the *back-end*, is responsible for generating a particular target-specific language from the intermediate language.

Several preliminary intermediate language formats were developed for the TTE in order to realise these benefits, as well as considering transformation directly to the Z Interface Format (ZIF) language specified in the Z base standard (Brien and Nicholls, 1992). However, it was observed that in all these cases the level of abstraction derived was similar to the ZAL language itself, and there were no real benefits to be gained from implementing an intermediate language that could not already be obtained from transforming directly to the ZAL language. This is a credit to the ZAL language itself, in that it is sufficiently abstract to be used as an intermediate language in its own right, whilst still retaining executable semantics. Ultimately, it was therefore decided to transform directly from the original Z specification to the ZAL language.

5.1.7 Accessibility of the Executable Representation

This debate arose from research into how specification developers actually *use* animation systems in a practical way. Much is made in the literature of that fact that the specification should always be the primary source of information to the system designers. However, the introduction of an animation phase based on a rapid prototype of the system, affords the possibility that the executable representation *itself* becomes the deliverable from the specification process.

From research into how users approach animation systems, it has been observed that once a specification is written and transformed into an executable representation, the *executable representation* itself tends to become the focus of refinement as the animation process proceeds, rather than the original specification. This may be because many specifiers are actually experienced developers, whose tendency is to refine the executable representation as problems are uncovered, in the same way as one would debug a

program. However, the danger is that the executable representation will diverge from the original specification and hence information and abstraction will be lost. In particular, as described later the nature of the transformation process inherently discards much of the original type information in the original Z specification, as we move from one semantic universe (i.e. Z) to another (i.e. ZAL). This is characteristic of the move from a logical representation to a computational representation, in that whilst we can map certain operational features, due to the gap in levels of abstraction it becomes increasingly difficult to map semantic information.

There are two approaches to overcoming the problem of animators refining the executable representation rather than the original specification: Firstly it could simply be accepted, and the necessity is then to provide a means of *re-engineering* the original Z specification from its executable representation. However a brief study of the executable subset of Z constructs implemented in ZAL for example, shows that this cannot easily be achieved. In particular we cannot re-create the *type* of the variables defined in the specification, as this information was discarded in the original forward transformation.

The alternative is to provide sufficient support in the TTE such that it becomes *unnecessary* to modify the executable representation. Thus in using the animation system, the specifier is more comfortable in referring back to the original Z specification, safe in the knowledge that the transformation process will faithfully reproduce a refinement *identified* in the animation environment, but *captured* in the original specification.

This places further emphasis on the requirements of the TTE, in that it becomes important for the TTE to support as much as the transformation process as possible (either automated or otherwise). Similarly, it must also generate *animation-supporting* features of the ZAL language via an independent mechanism, in order to reduce the need to modify the executable representation directly.

5.1.8 Ensuring Transformation Correctness

To be of use, the transformation process must be correct in some sense in transforming between the $LL(k)$ grammar derived for the Z notation discussed in section 4.2.7, and the grammar of the ZAL language developed for the TTE.

Firstly, it became clear early in the research of the transformation engine, that a true and correct transformation could not be guaranteed without accurate type information extracted from the original Z specification. Secondly, even though the animation environment is based in the imperative domain, it is not a compiler. Therefore, it cannot detect all semantic faults in the original Z specification.

The reasons for this are rooted in the use of a LISP environment for the animation engine. The data manipulated in LISP programs consists largely of lists and symbols, however there is nothing about LISP, which restricts us to manipulating only these objects (for example we can still write LISP programs to deal with numeric quantities in the same way as we can with 'C' or FORTRAN). However, LISP has a different approach to data types than one would find in an imperative programming language such as 'C'. It is convenient to think of the data types of LISP as a hierarchy, in which some of the data types are simply more specific versions of another. As Wilensky (1986) points out, the consequence of this is that it is meaningless in LISP to ask what the data type of an object is. For example, what is the data type in LISP of 3.2? In fact it could be *number*, *float*, *atom* or *single-float* depending upon the context in which it occurs. A consequence of this is that the LISP *type-of* operator could return any of these types depending upon the implementation. This functionality allows us to write very compact programs in LISP in which type information can be determined at run-time and dealt with appropriately. Indeed, this feature is actively exploited in ZAL in order to simplify the animation language and allow us to represent symbolically the abstract nature of Z data types. For example, the ZAL *eqz* operator will determine equality for expressions of any particular data type, and the ZAL *mks* operator will build sets of any particular data type.

However, this behaviour is diametrically opposed to the rigorous type system embodied by the Z notation. Therefore it is not possible for the ZAL animation environment to perform rigorous type checking, and indeed it is not intended for this purpose when considering that the objective of animation is to execute a rapid prototype rather than generate one.

Therefore, from the preceding discussion, in order to guarantee that the transformation process produces a deterministic and correct result, it is necessary to guarantee the *type correctness* of the input specification. The design decision resulting from this work is that in order to guarantee correct transformation, the transformation engine will produce no output whilst there are syntactic or semantic errors present in the original Z specification.

5.2 Realisation of the TranZit Transformation Engine

The essential function of the transformation engine is to provide a mapping between the operations and data types defined in the abstract Z universe, and the operations and data types available in the ZAL animation environment based on LISP. This is illustrated in Figure 5-4.

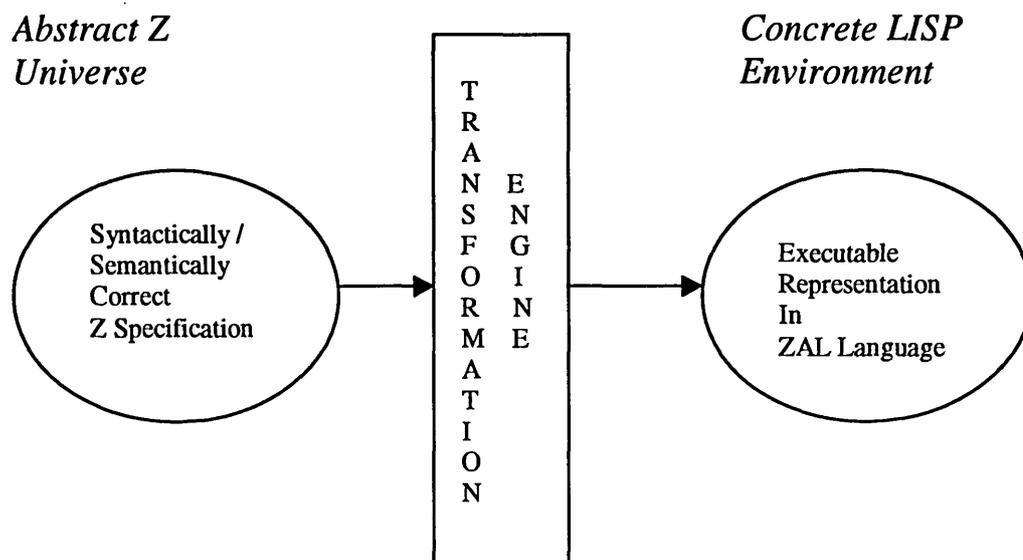


Figure 5-4: Mapping Between Semantic Universes

The starting point for this process is the $LL(k)$ Z notation grammar developed for this project as described in section 4.2.7, and the grammar of the ZAL language. The associated grammar definitions are given in Appendix I: $LL(k)$ Grammar for the Z Notation and Appendix II: Context-Free Grammar of the ZAL Language respectively.

We therefore seek to design a transformation engine that can achieve this mapping in a deterministic and consistent manner. The most obvious way of achieving this result is to define a set of re-writing rules or *productions* embedded within the $LL(k)$ grammar. Since the pre-condition has already been made that the Z specification to be transformed must adhere to the syntactic and semantic rules for the Z notation, it is possible to define a set of productions which will achieve the desired result in a deterministic fashion.

However, on examination of the grammars, it can be seen that there is not a one-to-one correspondence between the two. There are several reasons for this:

- Not all aspects of the Z notation grammar relate to executable components within the ZAL language. For example, the ZAL language is not concerned with schema variable declaration syntax, other than to note that an input or output variable of some type has been defined. The ZAL language grammar effectively represents an *executable subset* of Z operations, which are implemented within the animation environment.
- The grammar of the Z notation does not make explicit certain operational features required in the executable representation. For example, the section of the Z notation grammar, *Expression .. Expression*, denoting a sequence of expressions could represent a sequence of numbers or a function application. These two types have separate grammar representations in the ZAL language (e.g. the ZAL grammar of a function application requires the presence of the ZAL *applyz* keyword).

In essence this means that the production system must be supported by additional semantic information captured from the Z type system, in order to deduce which production to apply in any particular case.

5.2.1 TranZit Transformation System Architecture

The TranZit transformation Engine consists of a number of key components as indicated in Figure 5-5:

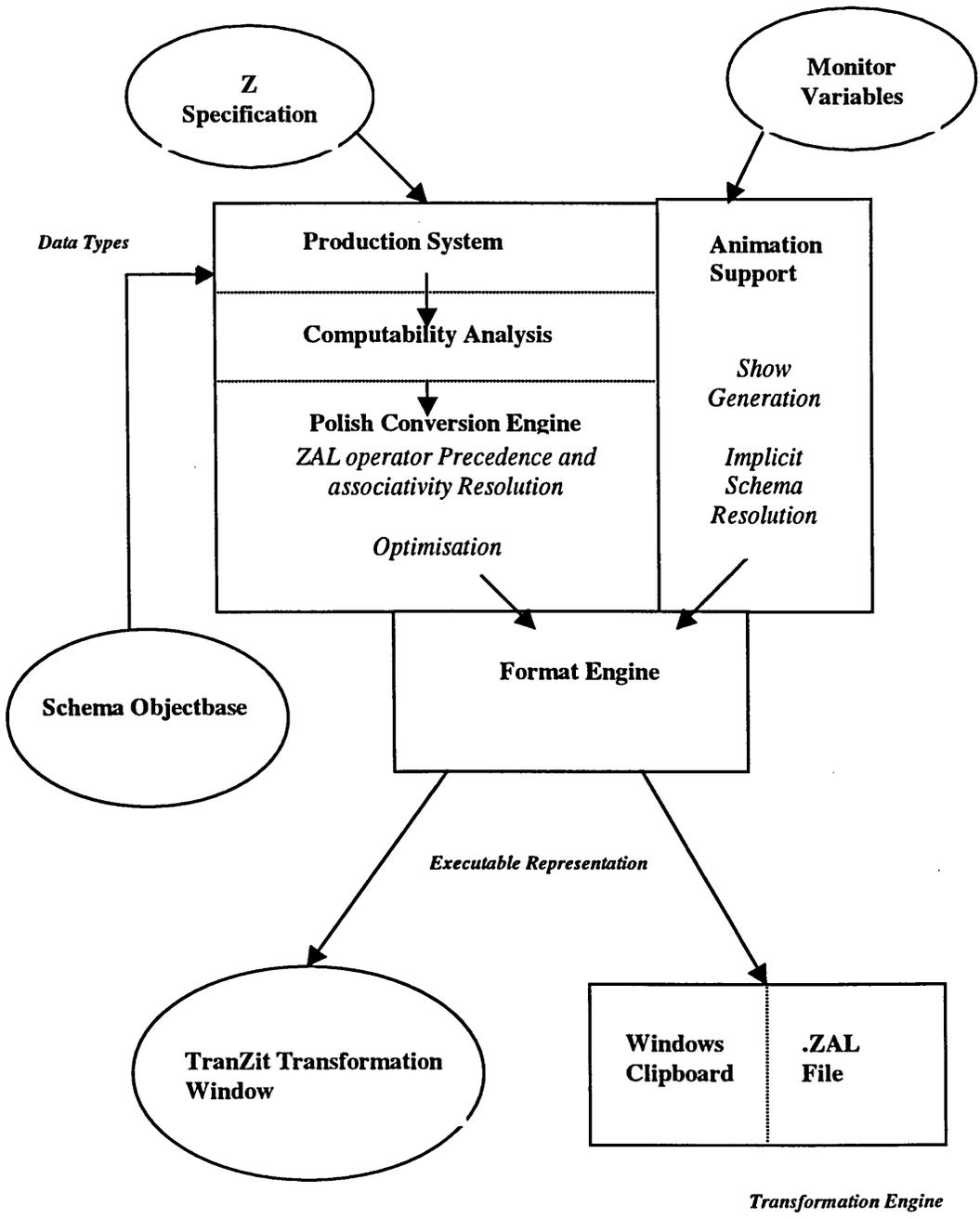


Figure 5-5: TranZit Transformation System Architecture

The individual components of the TTE are described below:

5.2.2 The Production System

The production system is implemented as a number of sub-components within the re-written LL(k) grammar forming the basis of the TAS parser. As the parser executes, components are invoked at particular levels within the original grammar, which correspond to executable transformations. Using the original Z notation grammar allows the production system to know the context of the parse at any point, and thus select the correct production to produce the desired executable representation in the ZAL language. However, it is often the case that the correct production cannot be selected without additional *type* information. Hence the production system components may also interrogate the schema objectbase described in 4.3.1, in order to determine the choice of executable representation. A good example is the case of a sequence of expressions, in which the production system must determine whether these are actually a sequence of individual identifiers, or a function application.

The output from the production system is a sequence of *operator* and *operand* tokens, which represent the basic components of the transformation. However, this sequence is in the same form as the associated Z notation (i.e. prefix, in-fix or post-fix representation). Similarly, the operator tokens have not been ordered according to the rules of *precedence* or *binding association*. This token sequence is therefore passed onto *Polish conversion engine* which will resolved these issues.

5.2.3 The Polish Conversion Engine

The function of the polish conversion engine is to reduce the sequence of operator and operand tokens produced by the production system to LISP *s-expressions*, inserting appropriate ZAL operator keywords or LISP functions for the operator tokens. The operator keywords are organised by table look-up, to allow for the fact that keywords may be changed.

This is one of the most complex operations within the transformation engine, as it involves reducing Z prefix, infix and post-fix operators to a common polish format (*operator arg₁... arg_n*). The problem is similar to that of a compiler which typically reduces a source program to an intermediate language consisting of a sequence of *n-tuples* that the code generator then converts to the target assembly language. In addition, this function must also take account of the *precedence* of Z operators in expressions, and also whether the *binding association* of the operators is *left*, *right* or *unary*. Finally, the function must perform *bracket balancing* to ensure brackets are inserted in the correct places and are opened and closed as a pair.

The polish conversion engine is also responsible for performing *optimisation* functions including aggregation of implicit connectives such as *and*, *or* and *imply*. For example:

```
(and
  (and
    (op1 arg1 arg2)
    (op2 arg1 arg2)))
```

Can be optimised as:

```
(and
  (op1 arg1 arg2)
  (op2 arg1 arg2))
```

This frequently occurs in the transformation of multiple predicates within the body of a schema, in which each of the predicates is implicitly *and*'ed together.

5.2.4 The Computability Analyser

The computability analyser is subordinate to the production system, and is intended to determine conditions in which automatic transformation of the original Z specification may not be possible. If this is the case, then the computability analyser invokes the *TranZit Transformation assistant* to elicit additional transformation information from the

user. A detailed explanation of this component of the transformation engine is deferred until the discussion of the non-computable aspects of the Z notation in section 5.3.

5.2.5 Animation Support

The animation support function supplies additional ZAL language components to the transformation process which support the animation environment directly, rather than being explicit components of the original Z specification.

Two important functions are handled by this component:

- Support for *shown* variables (i.e. variables associated with the ZAL **:SHOW** meta language described previously)
- Implicit Schema Resolution

5.2.6 Support for Shown Variables

In the ZAL language it is possible to specify that the animation environment output information to show the changes to particular variables as the animation executes. This concept is similar to a *watchpoint* in a conventional debugger. Such information is submitted to *TranZit* from the *tools->set show variable* dialog box as shown in Figure 5-6:

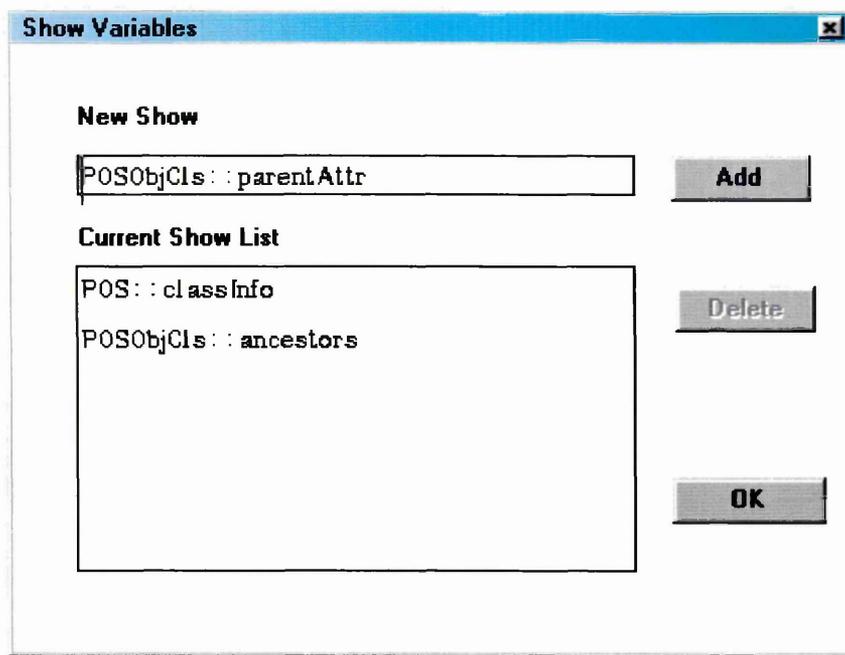


Figure 5-6: Set Show Variable Dialog

The user either *selects* a variable from the original specification, or manually enters show data of the form $\langle \text{schema ref} \rangle :: \langle \text{var name} \rangle$ into the edit control, to identify which variables the user is interested in monitoring as the animation progresses. Provided, this information can be resolved by the animation support function, the TTE will generate appropriate ZAL ‘:SHOW’ commands at the correct point in the output transformation.

5.2.7 Resolution of Implicit Schemas

This function deals with support for implicitly defined schemas generated by the use of the Δ and Ξ conventions in the original Z specification (Spivey, 1992).

As indicated by Spivey (1992), operations on data types are specified by schemas, which have two copies of state variables amongst their components; an *undecorated* set corresponding to their state before the operation, and a *decorated* set corresponding to their state after the operation. There is a convention that whenever a schema S is introduced into the state space of an abstract data type, the schema ΔS is *implicitly* defined as a combination of S and S' , unless another explicit definition for ΔS already exists in the specification. The implicit definition for ΔS is:

$$\Delta S \cong [S, S']$$

In a similar way, operations may wish to access information in an abstract data type, without changing its state in any way. Again the schema ΞS is *implicitly* defined whenever schema S is introduced into the state space of an abstract data type, with the implicit definition:

$$\Xi S \cong [\Delta S \mid \Theta S = \Theta S']$$

The user is free to use these conventions within Z specifications captured in *TranZit*, and the TAS understands the associated semantics. However, whilst the semantics of these conventions are clear within the context of the original specification, in the executable representation these conventions need to be made explicit. *TranZit* therefore includes an *implicit schema resolution system*, which maintains a database of schema definitions associated with these conventions. Once the transformation engine has completed a pass of the specification, if any implicit schema definitions remain unresolved, the TTE generates *explicit* executable representations of these schema automatically according to the definitions above. This ensures that when the executable representation is submitted to the ZAL animation environment, all schema references are explicitly resolved.

5.2.8 The Format Engine

The final component of the transformation process is the format engine. The format engine is controlled by information from the transformation system dialog box, as shown in Figure 5-7:

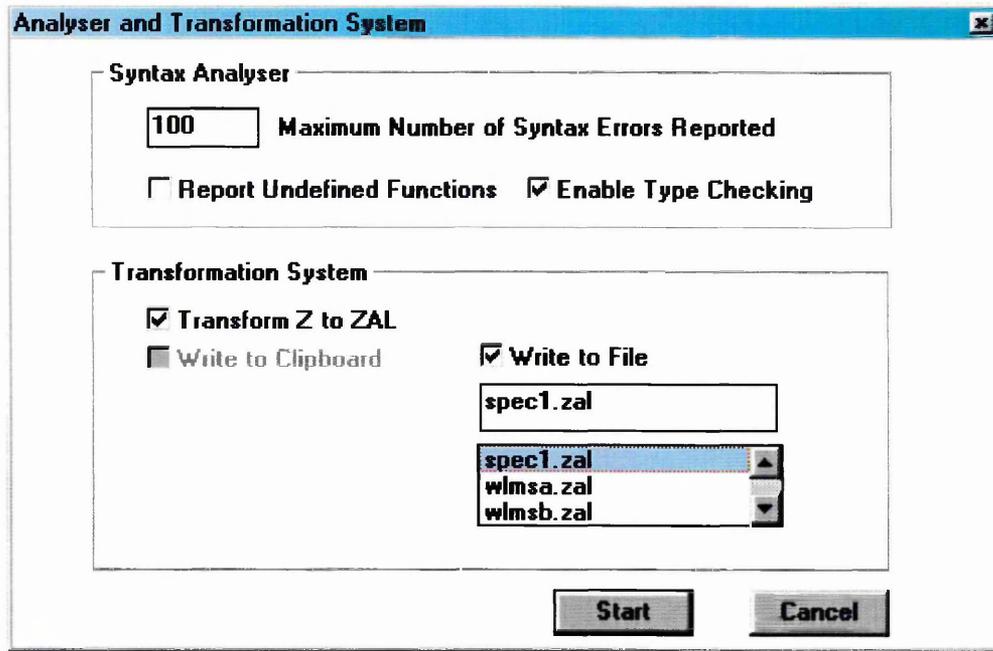


Figure 5-7: The Transformation System Dialog

The transformation system group in this dialog determines:

- Whether a transformation is to be produced (this also depends upon the success of the syntax and type check of the specification).
- Whether the output is to be written to a specified file, or to the Windows Clipboard for pasting into another application.

If the transformation is successful, the Format Engine invokes a *pretty-print* operation, and displays the transformation in a separate application window. This allows the user to audit the transformation before deciding whether to proceed to the animation environment. A typical window arrangement for transformation auditing shows the original Z specification and the corresponding transformation on the same screen, as shown in Figure 5-8:

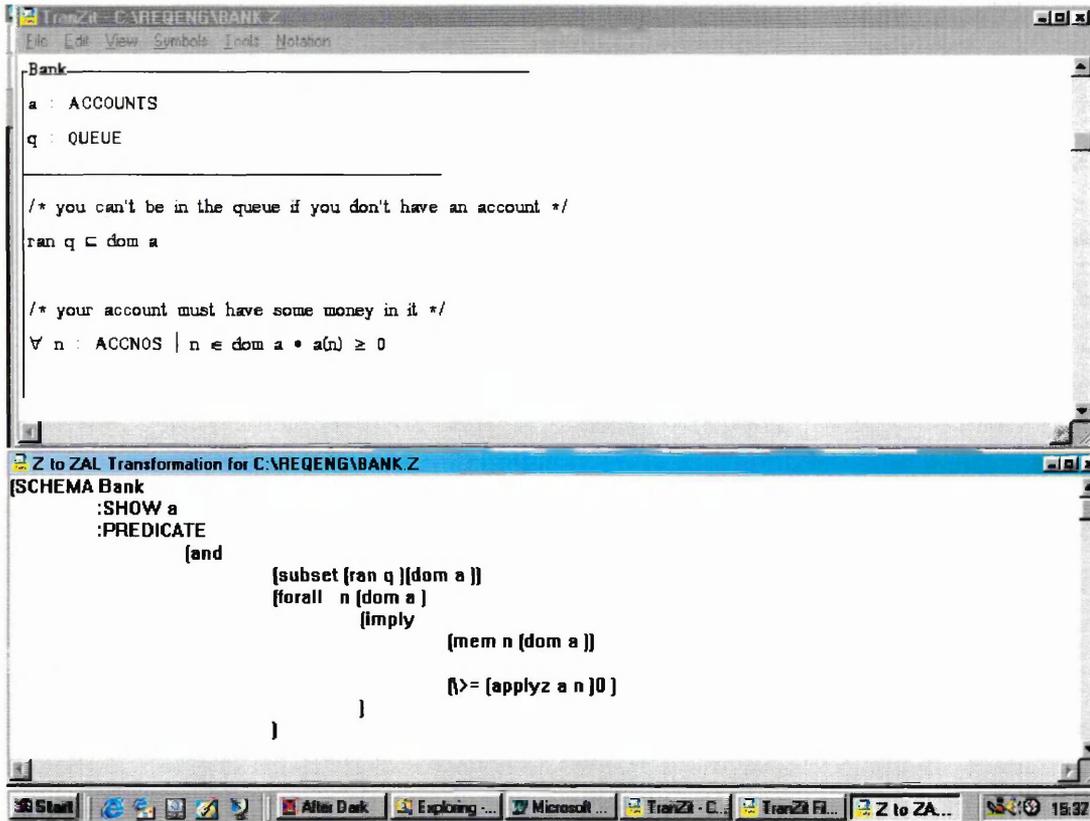


Figure 5-8: Typical TranZit Window Arrangement for Transformation Auditing

If the user decides to proceed to the ZAL animation environment, then the user can specify whether the transformation is stored on the Windows Clipboard, or saved to a particular file for importing into the ZAL environment.

5.3 Non-Computable Aspects of Specification Languages

Thus far the transformation process has been considered to be an automated conversion from the Z notation to the ZAL language. The tacit assumption has been made that every construct within the Z notation has a computable representation in the execution environment. However, this is not the case, and therefore the TranZit transformation engine requires a strategy to deal with the problem of non-computable clauses within Z specifications.

The issue of the *computability* of clauses in formal specifications has provoked many debates in research circles. Perhaps one of the most important papers in this area, which still provokes much discussion, is that of Hayes and Jones (1989). In this paper Hayes and Jones seem to argue against attempts to pursue research into the transformation of specifications into executable representations. They justify this view by asserting that the needs of the transformation process inevitably restrict the expressiveness of the specification language, and may influence the specifier in considering implementation issues rather than capturing pure requirements. Their paper is well supported by a detailed discussion of specification problems, which in their opinion cannot be made executable. In particular, the paper argues that one should be able to specify the well-known non-computable *halting problem* (Wulf *et al.*, 1981) in order that a single notation can be used to explore both the theoretical and practical applications of computing.

The view taken in this research, which is supported by others (Fuchs 1992, Valentine 1995, Dick *et al.* 1990, Knott and Krause, 1992), is that useful results can be obtained from transforming formal specifications into executable representations. Indeed, as discussed later, it is believed that the development of the TranZit transformation engine in conjunction with the ZAL language has shown that it is possible to produce executable representations at similar levels of abstraction to the original specification, whilst retaining almost identical structure.

This work also supports the view of Fuchs (1992) in that “.. *the lack of correctness of software is the most serious problem in software development, and not the possible lack of expressive power of the specification languages*”. Excluding the execution of specification languages needlessly deprives specification writers of a powerful tool for validating specifications against informal requirements. In addition it is believed that there is a strong influence to combine deductive and inductive approaches to software engineering in order to improve the understanding and application of formal methods in solving practical problems (Siddiqi *et al.*, 1998).

5.3.1 Computability and the Z Notation

The Z notation is non-executable. This is easily demonstrable by the fact that one may write a specification of the Halting problem (Wulf *et al.*, 1981) in Z. Particular elements of the Z notation which make it non-executable are well understood and are described in detail in Hayes and Jones paper (1989). In general the problem distils to the fact that the Z notation allows us to express concepts without regard for:

- *Finiteness*,
- *Efficiency*,
- *Determinism*.

The Von Neumann model of computation used in a physical computer system inherently imposes finiteness by the fact that the computer has a limited memory space. Thus, it is not possible to truly represent a basic concept such as \mathbb{N} (the set of positive integers including 0) in a computer system. This is because the representation used to store a number in the computer's memory is limited to some arbitrary value determined by the number of bits available in the basic machine word.

Computation also imposes efficiency requirements, by the fact that a computer cannot perform an infinite number of operations in a finite time. It is easy to specify so-called *NP-Complete* problems in Z (e.g. the Hamilton tour problem), for which executable solutions are known to exist, but which are *computationally infeasible* (Goldschlager and Lister, 1982). Thus an executable representation of some abstract problem may not generate a result in a reasonable time depending upon the number of data points entered into the computation.

Finally, computation imposes determinism by the fact that the Von Neumann model of computing is essentially one of sequential operations on the program state. In a logical specification this restriction is not enforced, thus the specification of an (incomplete) number sorting operation (taken from Hayes) may appear in Z as follows:

$$\text{Sort} \equiv [\text{Input?}, \text{output!} : \text{Seq } \mathbb{N} \mid$$
$$\text{IsOrdered}(\text{Output!}) \wedge \text{IsPermutation}(\text{Input?}, \text{Output!})]$$

The result of the *Sort* operation is a *conjunction* of two processes, the second of which generates a (possibly) infinite set of permutations of the input represented as sequences, and the first of which returns an infinite set of sequences which are numerically ordered. Depending upon the *order* in which these individual processes are executed, the result of executing the *conjunction* of these processes may or may not terminate. Thus it is necessary to *reason* about the problem in order to impose an execution strategy which will constrain each process output in some meaningful way, and therefore guarantee termination. Such specifications are said to exhibit *external non-determinism*.

It is also the case that *internal non-determinism* may exist, by the fact that a system's overall operation may be deterministic, but it may be composed of non-deterministic operations. Systems using parallelism are a case in point. For example, an operating system's paging mechanism may have a non-deterministic specification, but the result of running user programs on the system must be deterministic. Hayes and Jones argue that the ability to express non-determinism in specification languages is vital to ensure that the specification is not constrained by implementation detail.

5.3.2 A Strategy for Dealing With Non-Computable Clauses in Z

As highlighted in section 5.3.1, the problem of transforming an abstract specification to an executable representation is rooted in the disparity between the semantics of the logical domain in which the specification is expressed, and the Von Neumann model of computing imposed on the executable representation. Whilst a general automated solution to this problem is non-computable, it is possible to define a strategy for *partial* automated transformation coupled with *user assistance*, which can yield useful and practical results. This is the essence of the strategy used by *TranZit* in dealing with the problem of non-computable clauses in Z. An example, which highlights this strategy is described in section 5.3.3.

5.3.3 Example: IsAPerfectSquare

Non-computable problems can be identified in seemingly trivial specifications. For example, the boolean operation *is_a_perfect_square*, adapted from Hayes and Jones (1989), can be specified as:

$$\text{IsAPerfectSquare} \equiv [i? : \mathbb{N} \mid \exists j : \mathbb{N} \cdot i? = j^2]$$

In this case, the intention of the specification is to define an operation in which the integer *i?* is *tested* to determine whether it is a perfect square; i.e. does there exist some positive integer which is the square root of *i?*. A naive transformation of this operation into the ‘C’ programming language would appear as shown:

```
typedef BOOLEAN unsigned char;
BOOLEAN IsAPerfectSquare(unsigned int i)
{
    unsigned int j;

        j=0;
        while(i != (j * j))
            j = j + 1;
        return(TRUE);
}
```

As shown above, a naive executable representation of this operation would repeatedly enumerate values for *j* from the set \mathbb{N} (modelled by the ‘C’ data type *unsigned int*), before performing the test $i? = j^2$. If the test is *true*, then the function terminates, however if the value of *i?* is not a perfect square (e.g. 5), the function continues to loop internally *ad infinitum* (or more likely until an exception occurs when j^2 exceeds the maximum integer value which can be represented by the host computer).

Closer examination also reveals that a more subtle problem exists with the process of enumerating the variable *j*. When designing the program, the assumption has been made that values for *j* will be enumerated sequentially, starting from 0 and incrementing by one on each iteration. The programmer has therefore *reasoned* about the meaning of the

specification itself in order to deduce a strategy for enumeration, which he or she knows is likely to give a result in the executable representation. As a further example, consider the case of searching for a solution to the problem:

$$\exists j : \mathbb{N} \mid j < 10 \cdot (j + 3)/(j - 3) = 2$$

The solution is clearly $j = 9$, however a sequential enumeration strategy for j would stall at $j = 3$ with a *divide by zero* exception.

In general, this reasoning process may be quite complex and cannot be deduced from a static analysis of the specification by an automated transformation system. Again, the Halting problem tells us that this is the case since we cannot write a program which will identify whether an executable representation will terminate for some candidate input data.

5.3.4 Adding Constraints to Non-Computable Clauses

Many similar examples can be given involving the use of universal and existential quantifiers, which in general transform to a search of an infinite space. A similar problem also exists within a set comprehension of the form $\{D \mid P\} \cdot [E]$ in which the values of the set are implicitly *generated* by some (optional) general expression E constrained by an (optional) predicate P . For example, the following set comprehension describes the set of so-called *Hamming Numbers*, whose prime factors are either 2,3 or 5:

$$\{ x : \mathbb{N} \mid \forall y : \text{PRIMES} \cdot x \bmod y = 0 \Rightarrow y \in \{ 2,3,5\} \}$$

In this case, the set comprehension requires the selection of *enumerations* for x and y from the infinite sets \mathbb{N} and PRIMES respectively and then performs a test for set membership of x by a modulus operation and a further constraint on y . A naive executable representation of this set comprehension would be hopelessly inefficient, and in any case the set is unlikely to be computed in its entirety since the set of Hamming number is quite possibly infinite. However as Hayes and Jones point out, even though a

specification may contain clauses which are potentially non-computable, if these clauses are conjoined with additional constraints then the whole may be computable.

Exploring this idea further, it is clear from an examination of the naïve transformation of the *IsAPerfectSquare* example given in section 5.3.3 that there is a missing loop termination condition in the case of a result not being found. An obvious condition is that the loop should continue only whilst $j < i$, leading to the following optimised specification:

$$\text{IsAPerfectSquare} \cong [i? : \mathbb{N} \mid \exists j : \mathbb{N} \mid j < i \cdot i? = j^2]$$

With the corresponding naïve transformation in ‘C’,

```
typedef BOOLEAN unsigned int;
BOOLEAN IsAPerfectSquare(unsigned int i)
{
    unsigned int j;

    j=0;
    while((i != (j * j)) && (j < i))
        j = j + 1;

    return(TRUE);
}
```

However, there are still three problems with this naïve transformation. Firstly, as before, the transformation will only work if a sequential enumeration strategy is adopted for j beginning at 0. This is not indicated by the original specification and a different strategy may cause the executable representation to fail incorrectly if the constraint $j < i$ is broken before exploring all possible enumerations of j .

The second problem arises since the deduction of the new loop constraint $j < i$ can only be developed by reasoning about the problem, and therefore cannot be determined by an automated transformation system.

Thirdly, a new problem has now been introduced, in that if the loop terminates due to the loop constraint being broken, the result of the function is still *true*.

The solution to the first two problems lies in the recognition that the constraint imposed is a weaker condition over the quantified variable j than the general case of selecting enumerations for j from a *finite subset* of \mathbb{N} . However, this requires that the *human user of the animation* place a limit on the search space, as in general this process cannot be automated.

For example, for the purposes of animating the specification, it could be stipulated that the enumerations for j be taken from the set of integers between 0 and 10. The formal specification would then appear as:

$$IsAPerfectSquare \cong [i? : \mathbb{N} \mid \exists j : \mathbb{N} \mid j \in 0..10 \cdot i? = j^2]$$

With the corresponding naïve transformation in ‘C’:

```
typedef BOOLEAN unsigned int;
BOOLEAN IsAPerfectSquare(unsigned int i)
{
    unsigned int j;

        j=0;
        while((i != (j * j)) && (j <= 10))
            j = j + 1;

        return(TRUE);
}
```

However, the problem associated with the incorrect result *true* if the loop constraint is reached has still not been addressed. To provide a correct transformation it is required to introduce additional program elements to make explicit the return condition, as shown below:

```

typedef BOOLEAN unsigned int;
BOOLEAN IsAPerfectSquare(unsigned int i)
{
    unsigned int j;

        j=0;
        while((i != (j * j)) && (j <= 10))
            j = j + 1;

        if(i == (j * j))
            return(TRUE);
        else
            return(FALSE);
}

```

Whilst it is quite easy to show that this transformation will terminate with a correct result for any value of *i*, this transformation could no longer be considered naïve.

The solution to the final problem lies in the recognition that the use of an imperative programming language such as ‘C’ imposes additional transformation problems, due to the fact that the language itself does not inherently embody the notion of *truth* of a statement. This means it is necessary to introduce additional imperative program elements to model the semantics of the existential quantifier \exists itself, which are in general a function of either the constraint or the existentially quantified predicate (whichever may be the more efficient). This suggests it is necessary to employ an executable representation using a logic-based *declarative* programming language such as LISP or PROLOG, as this facilitates a more automated transformation. As an example, the automated transformation by TranZit of this revised specification into the corresponding ZAL language is shown below:

```

(exist j (mks 0 1 2 3 4 5 6 7 8 9 10 )
  (and
    (mem j (mks 0 1 2 3 4 5 6 7 8 9 10 ))
    (eqz i (* j j ))
  )
)

```

Several of these ideas are akin to those presented by Kowalski (1979) in his paper *Algorithm = Logic + Control*. In this paper, Kowalski explains the relationship between the *logical* representation of a problem domain, and the *control strategies* for implementing them. In particular Kowalski notes that different control strategies for the same logical representation may have different behaviour in terms of efficiency, as suggested by the examples above.

5.3.5 The Computability Analyser: Identifying Enumeration Functions

Whilst it is not possible to devise an automated general transformation of non-computable clauses in Z, it is possible to identify *specific* conditions within otherwise non-computable clauses, for which automated transformation is possible. Programs of this nature which accept specific conditions which are known to terminate, whilst rejecting some of the conditions which do not are termed *partial decision procedures* (Wulf *et al.* 1981).

The philosophy adopted by the TTE allows users to specify problems using the Z notation in whichever style best fits the semantics of the problem domain. At the point of transformation, the TTE will report constructs that are possibly not executable. This function is embodied within the *computability analyser* of the TTE and involves the *detection* of constraints termed *enumeration functions* within otherwise non-executable clauses.

Two classes of enumeration function can be identified:

- **Explicit:** For example, within clauses of the form:

$$\exists D \mid P \cdot Q, \forall D \mid P \cdot Q \text{ or } \{D \mid P \cdot E\}$$

Here predicate *P* explicitly constrains the search space of variables introduced by declaration *D*, or

- **Implicit:** By the fact that for the specification to be of practical use, the concrete data structures of the corresponding animation must be populated with finite candidate data.

Thus, the *enumeration function* is inherently related to the execution strategy of the underlying animation system. In particular, the transformation engine makes the tacit assumption that values for the enumeration will be generated in a deterministic fashion.

For example, consider the case of a universal or existential quantification of the form:

$$\exists D \mid P \cdot Q, \text{ or } \forall D \mid P \cdot Q$$

Here, the computability analyser expects to deduce an *enumeration function* to generate values for the constraining variable introduced by P . To meet the requirements for *explicit enumeration*, the computability analyser requires that predicate P has the type signature:

$$P : X \times PX \quad ; \text{ where } X \text{ is a basic type.}$$

If $P \cong x \in \text{dom}(y)$, where $x: X$ and $y: P(X \times Z)$, or $P \cong x: \mathbb{N} \mid x \in 1..100$, then this criteria is satisfied, whereas if $P \cong i, x: \mathbb{N} \mid x < i$ then the criteria is not.

It could be argued that even though the explicit enumeration criteria is satisfied, this may still *logically* result in a search of an infinite set. For example, consider the following specification for a system invariant modelling a banking system in which accounts must remain in credit:

ACCNOS == \mathbb{N}_1

MONEY == \mathbb{N}

<p>Bank</p> <p>$accounts : ACCNOS \rightarrow MONEY$</p> <p>$\forall n : \mathbb{N}_1 \mid n \in \text{dom}(accounts) \cdot accounts(n) > 0$</p>

This specification chooses to represent customer account numbers (modelled by ACCNOS), as strictly positive integers. The enumeration function $\text{dom}(\text{accounts})$ therefore *logically* represents a possibly infinite set of positive numbers. However, to be of any *practical* use, the user of the animation must provide candidate data for the function *accounts*. This example therefore meets the requirements for *implicit enumeration*, and the corresponding automated transformation will terminate, when a finite candidate data set for the *accounts* function is assigned by the user at animation time.

The detection of an enumeration function by the computability analyser effectively ensures the property of *finiteness* in the resulting transformation, and allows the TTE to proceed with an automated transformation. Failure to detect an appropriate enumeration function requires *manual human intervention* to supply an appropriate constraint to allow the transformation to proceed. If the computability analyser determines that it cannot deduce an appropriate explicit or implicit *enumeration function* for a clause, then it invokes the Transformation Assistant described in section 5.3.6.

5.3.6 An Eclectic Strategy: The TranZit Transformation Assistant

The transformation strategy adopted by TranZit is eclectic in nature, drawing inspiration from the unobtainable yet idealised goal of automated transformation of the original Z notation, coupled with practical user involvement to resolve issues that are non-computable.

It is important to note that with the addition of finite set constraints, *TranZit* is able to perform the transformation of many specifications, such as those described in section 5.3.4, without any assistance from the user. However, it is considered unreasonable for the specifier to be *forced* into introducing appropriate executable constraints in a specification, simply to be able to use the TranZit transformation engine. To support this philosophy, *TranZit* embodies the concept of a *Transformation Assistant*.

If, during the transformation process, the computability analyser determines that the explicit or implicit enumeration criteria defined in section 5.3.5 are not satisfied, then in

order to proceed with the transformation *TranZit* must solicit help from the user. For example, in the following universal quantification, the computability analyser is unable to deduce either an explicit or implicit enumeration function for the constraining variable n , since the constraint represents an infinite set of integer values greater than 5.

$$\forall n : \mathbb{N} \mid n > 5 \cdot (a(n) - b(n)) > 0$$

TranZit therefore requests help from the human user by invoking the *Transformation Assistant* dialog box shown in Figure 5-9 below, and populating it with information concerning the associated schema and local variable that cannot be enumerated. The user is then prompted to manually enter an enumeration function in the ZAL language. In this case, an appropriate response *might* be **(dom a)**.

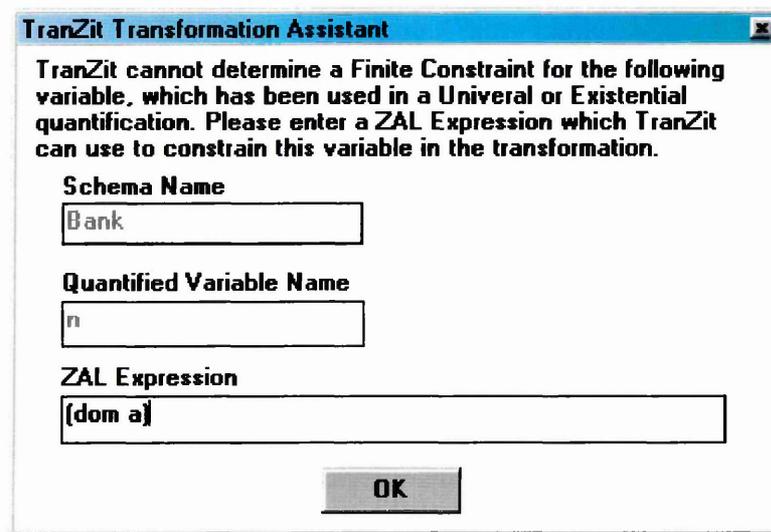


Figure 5-9: TranZit Transformation Assistant Dialog

TranZit then completes the transformation automatically as follows:

```
(forall  n (dom a)
  (imply
    (\> n 5 )
    (\> (- (applyz a n ) (applyz b n)) 0 )
  )
)
```

One could legitimately ask the question under what conditions the strategy adopted by *TranZit* will fail. An obvious example is the consideration of cases in which neither *TranZit* nor the human user can provide an enumeration function. Such possibilities might arise where there is a need to generate a *random* enumeration function for example. In this case, the user cannot supply the *TranZit* transformation assistant with a static ZAL expression that will achieve this goal. The problem lies in the recognition that the concept of randomness lies in the concrete rather than the abstract domain. To generate such values it is therefore required to fall back to the underlying execution mechanism of the animation environment, in this case LISP. To solve the problem, the user of the animation would need to supply some function in LISP to perform the random generation of values, and manually link this with the transformed representation of the original specification. Whilst this is possible to achieve with the current system, it goes against the ethos of the strategy adopted. In any case, it is extremely difficult to conceive of practical examples where this problem would arise.

Whilst it is not a failure condition, there are cases in which the strategy adopted will cause the computability analyser to err on the side of caution and invoke the *TranZit* Transformation Assistant when human reasoning suggests it is obvious how to resolve the lack of an enumeration function. For example, if we consider the following specification that generates all possible sub-sequences of some sequence s , it is clear that this involves infinite iteration over the infinite set $seq \mathbb{N}$.

$$s : seq \mathbb{N}$$

$$\exists x, y : seq \mathbb{N} \cdot x \hat{\ } y = s$$

From the previous definitions, the computability analyser cannot find either an explicit or implicit enumeration functions for the variables x and y . However, for any animation to be useful, the user would be expected to supply a value for s . Based on this assumption, were the computability analyser to extend its analysis to the *quantified predicate*, it could in principle deduce from the form of the predicate that x and y must be sub-sequences of s . It could then effectively internally *re-write* the specification as;

$$\exists x, y : SubSeqS \cdot x \hat{=} y = s$$

Where *SubSeqS* is the set of all sub-sequences of *s*. The re-written specification now meets the requirements for an *implicit* enumeration function in ZAL of (*powerset s*).

Whilst in principle, this would appear to be a suitable extension to the strategy adopted, it requires further research, especially as there is the possibility that the quantified predicate may itself contain non-computable elements. Similar approaches based on automated re-writing of specifications to afford executability have been suggested by Hörcher (1994).

5.4 Summary

This chapter has completed the description of the research and development of the *TranZit* system by examining the *TranZit* Transformation Engine. This is a critical system component as it provides the important link in the REALiZE process that enables the user to perform specification verification by animation.

The development of the *TranZit* transformation engine makes innovative use of ideas adopted from traditional compiler design coupled with novel techniques and strategies to overcome problems associated with non-computable aspects of the *Z* notation.

In particular, the strategy adopted is an eclectic approach, which aims to automate the process of transformation, in so far as is possible, coupled with user assistance on detection of certain non-computable conditions. These conditions have been characterised using the novel concept of explicit and implicit enumeration functions, which seek to constrain the scope of infinite objects. To resolve conditions in which automated transformation is not possible, the user is required to supplement the transformation process with expert knowledge gained from an understanding of the specification problem domain, which in general cannot be determined by static analysis of the specification by computer algorithm.

In general, it is considered quite natural to require the user to constrain the search space of an infinite set if the transformation is to be of any practical use, since there are very few practical applications which make use of the concept of infinity. Whilst this strategy cannot produce a *functioning* executable representation of the halting problem from its original Z specification for example, it has been found to be applicable in many practical specification problems.

It is admitted that the strategy implemented cannot mitigate all the problems associated with some of the contrived, theoretical specifications highlighted by Hayes and Jones (1989). However, it has been found that by the application of the transformation strategy described, the majority of specifications that have a practical application do not exhibit non-computable problems during animation.

In the next chapter this claim is supported by the exploration of transformation case studies, together with results obtained from the *real-world* use of the *TranZit* system by students and staff at SHU.

6. TranZit System Testing, Evaluation and Case Studies

In earlier chapters a number of fundamental considerations have been discussed which have guided the course of this work. In particular, the evolution of requirements engineering itself has been described, and the impact it has had on modern systems development has been assessed. The need for a requirements engineering process has also been identified and the subsequent development of the REALiZE process provides the foundation of a methodology upon which to capture a complete and unambiguous statement of requirements. To support this process the *TranZit* tool has been researched and developed to provide a powerful, integrated tool to support capture of specifications in the Z notation and transformation of such specifications into executable representations in the ZAL language for the purposes of *validation by execution*.

This chapter aims to;

- Demonstrate the quality of the TranZit software product by describing the testing strategy employed,
- Assess the *usability* of the system design from the user's perspective, and
- Evaluate what has been achieved by comparison with existing requirements engineering tools and by describing the capabilities of *TranZit* in assisting to solve practical requirements engineering problems. In the latter case, this is demonstrated by the description of two transformation and animation case studies, which aim to highlight the practical application of the *TranZit* tool in addressing real specification problems.

6.1 The Testing of TranZit

An important aspect of any software development process is the validation of the system by identification of a test strategy designed to eliminate the maximum number of errors in an application before it is released as a final product. *TranZit* itself was not formally specified, as is the nature of many exploratory developments, and hence it is not possible to perform mathematical verification to prove the program correctness. For an excellent text on this approach see Gries (1981). The validation task is therefore addressed by

more traditional methods of program testing. However, by careful selection of a test strategy that fits the problem domain, confidence can be gained that serious errors in the program have been eliminated.

TranZit has been implemented mainly in the 'C' programming language (Kernighan and Ritchie, 1978), and the research programme described in Chapter 2, has naturally led to a three-phase development, each of which was independently tested. This in turn gave confidence in each stage of the development before the next phase was implemented. The three development phases were as follows:

- Development of the *TranZit* Editor Sub-system (involving the windowing and menu system, file system interface and outline Schema Objectbase)
- Development of the *TranZit* Analyser sub-system (involving the lexical analyser, LL(k) parser implementation, detailed schema objectbase methods, error handling and type checker).
- Development of the *TranZit* transformation engine (involving mapping of executable constructs to the ZAL language, modifications to the parser for ZAL language generation, polish expression conversion, the precedence and binding association engine and finally the *TranZit* Transformation Assistant and animation support functions).

TranZit has been tested at each stage using a variety of well-proven approaches to system testing such as white-box *unit testing*, black box *integration testing* and techniques such as logic coverage, equivalence partitioning and boundary condition analysis (Pressman, 1982). In addition, each release has been *acceptance tested* or *alpha-trialled*, involving exposure of the system to a selected number of users who have exercised various scenarios and fed back problems.

Following alpha-trial, *TranZit* has then been released to the academic community at SHU, where it has been exposed to numerous specification problems on a number of different host platforms and machines.

The following describes the major results of the testing process.

6.1.1 Unit Testing

Unit testing focuses on the smallest unit of software design, i.e. the function. Unit testing is concerned with validating:

- The function interfaces
- The *major* logical paths in the function
- Data structure accesses
- Handling of Error and exception conditions

A unit testing strategy relies on detailed knowledge of the internal structure of a function in order to derive appropriate test cases. Hence unit testing is often referred to as *white-box* testing.

It is important to recognise that the scope of unit testing can be quite exhaustive, especially when one considers the number of possible execution paths through even a relatively simple function. The ease of testing is also affected by the program style, in that functions exhibiting poor internal cohesion and close coupling will complicate the testing process. It is therefore necessary to make a judgement based on experience, concerning the application of unit testing and depth to which it will be pursued.

In the *TranZit* system, it became apparent quite quickly that unit testing of the TAS was not appropriate. In particular the functions of the parser cannot be stimulated easily in isolation due to the nature of the program, and even if they could the likelihood of discovering a problem is extremely small due to the fact that errors in this type of program are essentially context dependent. Unit testing has therefore focused mainly on the *TranZit* Editor sub-system.

The *TranZit* editor sub-system can be unit tested using standard techniques. Because each function of the editor subsystem is inherently modularised by the fact that it must integrate with the Windows API, it is easy to stimulate operations individually and determine test cases to validate their operation. For example, unit testing of individual

dialog boxes is easily accomplished by exercising each control element of the dialog box independently (e.g. edit control, radio button, list box, e.t.c.). General editor functions can also be stimulated in the same way, by selecting menu items in turn.

The only major problem which was identified by unit testing of the editor sub-system was the interaction between functions performing manipulation of complete schemas (e.g. the cut and paste of an entire schema), and functions controlling the schema graphic outline. The user cannot normally access the schema graphic outline, as it is internally controlled by the *TranZit* program. However, errors were identified whereby it was possible to generate *gaps* in the schema graphic, which could not then be repaired. These problems were eventually traced to methods updating the schema objectbase incorrectly.

6.1.2 Integration Testing

Integration testing is concerned with systematically assembling software components whilst testing the *interface* between components as integration progresses. Common approaches to achieving this are *top-down* integration testing, in which high-level control modules are integrated first supported by *stub functions* that emulate the interface of subordinate functions. Conversely, *bottom-up* integration testing may be applicable in which sub-ordinate functions are integrated first supported by high-level driver programs typically mimicking some test case. These strategies are illustrated in Figure 6-1.

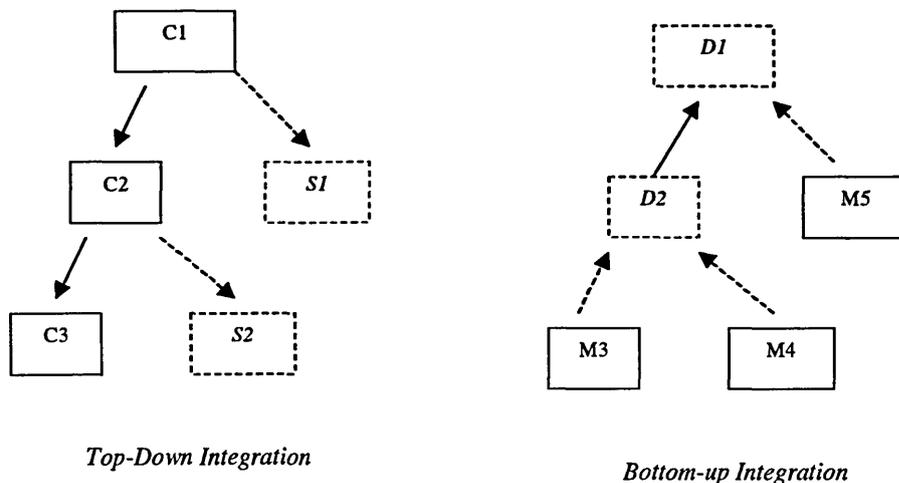


Figure 6-1: Integration Test Strategies

The nature of the parser and associated type checking functions, necessitated a *top-down* approach to integration test, since the parser is implemented using a technique termed *recursive descent* which inherently works from the general to the particular. Since *unit testing* of the parser functions could not be performed effectively, the parser has been implemented with built-in debug support. This is controlled by a simple compiler definition, and when enabled causes each function of the parser to output information to the Windows debug window associated with the code path and data elements it is selecting. It is therefore possible to *log* the execution path through the parser for individual test cases, which provides valuable information in detecting and rectifying errors and also provides a means of *regression* testing (i.e. testing that system modifications do not invalidate previously tested operations).

Another important design feature of the TAS assisting testing is that the type checker and transformation engine elements can be independently enabled, even though these are subordinate functions to the parser. This allows integration testing of the parser to be completed in isolation from the type checker and transformation engine. The integration test strategy is therefore based on proving the parser, then the type checker and finally the transformation engine.

Integration test case generation is essentially based on capturing sample elements of Z in the editor, which will exercise all of the LHS non-terminals in the grammar. Since there is a one-to-one correspondence between LHS non-terminals and functions in the parser, we can be sure that this strategy covers all the program components of the parser.

Once the parser was successfully tested, the type checker was enabled and the same set of sample Z elements was then used to provide a means of *regression testing*. Errors were rectified, and the sample Z elements modified to exercise different *types*.

Only when the type checker finished testing, and the TAS had successfully completed regression testing, was the development of the transformation engine instigated.

The transformation system was integration tested firstly by regression testing using the set of Z elements originally used to test the TAS, and then by defining test cases supplemented by a set of hand-crafted transformations which could be compared with the output of the transformation engine. Loading transformed specifications into the ZAL animation environment and exercising them using animation test cases then validated the transformation engine output.

Surprisingly, the major problems encountered in testing the TAS and TTE concerned memory management rather than errors in the parsing and transformation process. Unfortunately C and C++ do not inherently support *garbage collection*, making it the responsibility of the programmer to make sure that all allocated dynamic memory is explicitly discarded. Within recursive programs like the parser, type checker and transformation engine it is extremely difficult to track where memory is allocated and discarded. This leads to errors such as *memory leaks* (in which memory is not discarded by the program once it has finished with it) and *multiple deletions* (in which the same memory block is discarded more than once). Unfortunately, the Windows OS does not seem to care about this until a significant amount of local and global memory is lost and the PC performance begins to degrade, as system resources become scarce. Memory management errors are therefore difficult to identify and even more difficult to rectify. It was found that the only way to isolate errors in memory management was to change the basic *allocate* and *de-allocate* functions to uniquely *tag* each memory block requested and deleted by the program. This then allowed a tracking mechanism to be implemented thereby identifying situations in which functions of the TAS or TTE were incorrectly using dynamic memory.

6.1.3 Acceptance Testing

A simple definition of acceptance testing is that the test is deemed to have passed when the software performs in a manner that meets the user's requirements. Formal user tests are therefore devised to show *conformity* with the original specification of requirements, which are often supplemented by a *test-drive* or *alpha-trial* of the system in which users exercise the system in its *normal environment*.

It is not possible to perform a formal acceptance test of the *TranZit* product, since the development has been exploratory in nature. However, it is possible to check that the set of *product requirements* discussed previously has been met. These are contained in the checklist shown in Table 6.

Product Requirements	Requirement Satisfied
⇒ Full-screen WYSIWYG editor, combining mouse and keyboard input	✓
⇒ Support for the Z notation character set	✓
⇒ Automatic generation of notation graphics (e.g. schema outlines)	✓
⇒ Support for standard editor functions:	✓
⇒ Cut, paste, insert, delete, select, search and line goto, with appropriate notation support (i.e. cut, paste, delete of complete schemas)	✓
⇒ Ability to load and save work to hard disk. floppy drive.	✓
⇒ Ability to print specification on standard printers	✓
⇒ Support interworking with other specification documentation packages (e.g. MS Word).	✓
⇒ Provide automated support for the generation of schema components within the specification	✓
⇒ Provide automated support for the semantics of schema inclusion and schema hiding	✓
⇒ Provide automated support for type generation and checking within the Z notation specification.	✓
⇒ Provide Automated tools for syntax checking of Z notation specification	✓
⇒ Ability to transform the specification from its non-executable form, to an executable representation, suitable for validation in an animation environment.).	✓

Table 6: Product Requirements Checklist

It can therefore be shown that the *TranZit* product has met all the requirements originally defined.

In terms of alpha-trial, very few problems have been identified by users of *TranZit*, and the current version is in use by many undergraduate and post graduate activities within the computing department at SHU. In particular it is actively used by undergraduates as part of their coursework associated with the teaching of Formal Methods. *TranZit* has therefore been trialled in a wide variety of applications, and there is a high degree of confidence in the validity of *TranZit* as a stable product.

6.2 Assessment of the Usability of TranZit

Whilst the research associated with defining a set of product requirements for the *TranZit* tool gives a good deal of confidence in the features *required* of the tool, it does not validate that these features have been implemented in the most *usable* way from the user's perspective. In order to quantify this and to provide important feedback on improvements and possible future development directions for *TranZit*, a User Questionnaire was produced to elicit the opinions of people who actually use the system on a day-to-day basis.

6.2.1 Analysis of Feedback from User Questionnaires

The questionnaire is designed in four sections presenting questions covering general population information, requirements engineering, the Z notation and the *TranZit* Tool specifically. Each section also contains control questions to identify *noise* such as conflicting views and lack of understanding in order to validate the opinions given. These questions do not contribute directly to the results.

The questionnaire was circulated to a variety of people in the academic computing community at SHU, with differing backgrounds and experience of formal specification and CASE tools. Whilst some staff and postgraduates completed the questionnaire, the majority of results were obtained from final year Computer Science undergraduates using *TranZit* as part of their Formal Methods coursework. It should be noted that this group did not have a wide experience of using the *TranZit* transformation system.

In total, around fifty questionnaires were returned, and the results are shown graphically in Table 14 to Table 25 as part of Appendix IV: *TranZit* User Questionnaire Results. The results suggest general trends as follows:

The first section concerning population statistics, suggests that the questionnaire was answered mainly by people with a fair degree of experience in writing formal specifications and using the Z notation. A pleasing result is the higher than average experience of MS windows amongst the population, which increases confidence in the results concerning the *TranZit* user interface.

Section two of the questionnaire solicits general opinions on requirements engineering. There is strong agreement that requirements engineering is a key process in the software development lifecycle, and that producing a quality specification of requirements is very important before design commences. However, there is a strong opinion that specification writing is not easy, and the expectation is that the system requirements will change as the development progresses. There is also support for the use of formal specification, as well as strong support for computer-based tools to assist in this task.

Section three solicits opinions concerning the Z notation. Respondents believed that the Z notation is flexible enough to capture a wide variety of specification problems, and that Z specifications can be modified without too much difficulty. However there is a strong opinion that the syntax and type systems of Z are difficult to assimilate, and that Z notation specifications are difficult to read and understand. This confirms the view that rapid prototyping of Z specifications is important in increasing both specifier and customer understanding.

Section four solicits opinions concerning the *TranZit* tool itself. The general opinion is that the *TranZit* GUI is easy to use and that *TranZit* presents information in a well-structured and logical fashion. However, there is also the suggestion that *TranZit* could be improved in assisting users to resolve syntax and type errors generated by the TAS. Users are also divided about whether using *TranZit* increases their understanding of Z. In general there is strong agreement that *TranZit* is easy to use, and that the most useful feature implemented in the TAS. However, the editor GUI is criticised, and on further investigation it was found that this was related to the way in which Z notation characters are currently accessed. In general, users did not want to use the *power-user* Windows

accelerator keys, and felt that the alternative mechanism of access via the menu system was too cumbersome. An intermediate access mechanism, perhaps consisting of a floating toolbar, should therefore be considered in any enhancements to the system.

The results of this questionnaire are useful in both validating the design decisions made in the development of *TranZit* and identifying possible future enhancements to the system. They also reveal some interesting opinions regarding the Z notation and the use of formal specification, which highlight the need for computer-based tools to assist in the requirements engineering task.

6.3 Comparison of TranZit and other Requirements Engineering Tools

It is worthy of note that there are many other requirements engineering toolsets available, each of which address different elements of the requirements engineering task. In order to compare the work presented herein against other requirements engineering toolsets, Appendix III : A Review of Current Requirements Engineering Toolsets, contains a review of the *state-of-the-art* in requirements engineering tools based on formal methods, and a comparison with the capabilities of the REALIZE toolset, particularly *TranZit*. In particular, this review forms a basis to evaluate the unique characteristics that the *TranZit* tool possesses, and thereby evaluate what has been achieved by the project as discussed below.

6.3.1 The Use of Formalism

It is instructive to consider the formalisms that have been adopted by tools providers. Out of the twenty-four tools listed, only fourteen use a recognised, established formal specification language or notation such as Z or VDM-SL. The remaining tools have based themselves on proprietary extensions to these languages (such as VDM++), or developed languages specifically for the particular toolset. This trend has become particularly noticeable in recent years, as tools providers seek to overcome the problems of working with abstract formal notations such as Z, by developing specification language whose semantics are more amenable to computer-based manipulation.

This apparent dichotomy between the desire to use an abstract specification language and the need to be able to implement tools based on standard models of computing, has generated a large amount of debate in the research community. Much of this debate culminated in Hayes and Jones paper “Specifications are not (necessarily) executable” (Hayes and Jones, 1989), which in turn was answered by Fuchs (1992) in his paper “Specifications are (preferably) executable”. Since then, the research community has been divided over the merits of developing computer-based animation and proof engines based on formal notations such as Z and VDM.

There are good arguments for and against using abstract specification languages as a basis for formal methods tools, however the work presented herein is not as partisan as to believe that only one approach is sound. In any case the development of *TranZit* has demonstrated, as discussed later, that a useful contribution to the requirements engineering task can be achieved with a formal methods approach.

6.3.2 Tools Platforms

It is also interesting to compare the computing platforms upon which providers have chosen to base their tools. The REALiZE toolset is based on the Windows environment for a number of key reasons:

- The Windows environment is now the most popular desktop development environment for industrial applications.
- Windows has matured significantly with the release of Windows 95, 98 and NT, and now offers a whole host of development facilities at a fraction of the cost of those previously found on Mini computers such as Sun Sparc. This makes Windows-based PCs highly attractive to development organisations wishing to provide highly integrated facilities, without the need to pay for a costly infrastructure.
- The Windows GUI is now well accepted and understood by the vast majority of computer professionals, making it easily accessible to a wide range of potential users.

However, only six of the tools listed in the review make use of Windows as their primary environment. By far the most popular environment is UNIX on SUN Sparc machines. It is believed that this is a legacy from the fact that the development environment on UNIX was historically superior to Windows, and therefore many such machines existed in

academic institutions. However, this is no longer the case. The Windows API is now just as powerful as X-Windows for example, making it possible to implement high quality GUI's and applications on Windows machines, using modern languages such as C++ and Java.

6.3.3 Comparison of the TranZit and Formaliser Tools

In terms of GUI look and feel, the tool that comes closest to TranZit is *Formaliser* (Logica Inc, 1995), produced as a commercial Windows application by Logica Inc. Like *TranZit*, *Formaliser* offers Z specification construction and checking facilities, based on a WYSIWYG GUI. However, *Formaliser* uses syntax-directed editing, whereas *TranZit* implements an off-line syntax and type checker. As discussed previously, this was a conscious decision made on the basis of the research into formal specification construction techniques. It is believed that the approach adopted by *TranZit* is more accessible to first time specification writers, and in particular it allows specifiers to capture the essence of what they want without the need to be immediately concerned with the complicated syntax and semantics of Z.

An important distinction to be drawn is that *TranZit* can be used either as a stand-alone formal specification construction tool, or as part of an integrated animation environment. *TranZit* is also unique in the sense that it includes a novel, automated Transformation engine capable of transforming specifications captured in the Z notation into the ZAL language for the purposes of animation. As far as is known, this makes *TranZit* and the REALiZE toolset, the only toolset currently available which offers integrated specification construction and animation support for the Z notation on the Windows platform.

6.3.4 Comparison of the TranZit and ZFDSS Tools

Several tool reviewed have addressed the issue of transforming a formal specification into an executable language for various purposes, including validation. In this respect it is interesting to compare *TranZit* with the *ZFDSS* Tool (Zin, 1993) developed as part of the Ceilidh System (Zin and Foxley, 1991). To this end the different approaches taken in

preparing, checking and transforming the specification to an executable representation are explored.

The ZFDSS system is aimed at providing support for preparing, validating and refining formal specifications written in the Z notation, together with a mechanism for assessing their quality. In common with TranZit it is based on the so-called “liberal” approach to the application of formal methods (Nicholls, 1991), in which formalisation is used only when necessary or appropriate in the software lifecycle. In this way, not all the system components may be formally defined, and it may be considered unnecessary to *verify* the specification using formal reasoning.

In contrast to TranZit, ZFDSS is hosted in the UNIX environment rather than Windows™, essentially to make use of the large number of standard support tools provided within the UNIX development environment. Whilst Windows™ does not provide such a mature development environment, it has become much more popular for industrial applications development in recent years, hence the decision to host TranZit on Windows™.

A major design decision driven by the use of the UNIX platform is that Z specification *construction* in ZFDSS is based on the use of standard UNIX text editors, supported by the UNIX *roff* documentation preparation tool. In *roff* the input document consists of plain text and markers which define the expected output format of the document. Since Z specifications involve a high degree of graphical information, it is difficult for the user to visualise how the specification will look on paper using this input mechanism. ZFDSS therefore includes a Z pre-processor “*zpp*” for the purposes of converting mathematical passages into proper Z documents, supported by special markers representing Z mathematical objects. This is in contrast to the *WYSIWYG* editor approach adopted by TranZit in which the user is able to view the specification development directly in the editor, supported by a number of *language-aware* features which assist in the construction and maintenance of the Z specification document. It is believed that the *WYSIWYG* approach offered by TranZit is far more approachable to users, and supports the design goal of focusing the user’s attention on constructing the specification itself,

rather than the mechanics of accessing and drawing the graphical components of the Z notation.

In respect of checking and producing an executable version of the input specification, ZFDSS makes use of a so-called *conceptual model* of a Z specification. In essence this is nothing more than a set of references to specification objects in the source document, together with a set of relationships between variables and schemas. This is produced from the Z specification document by the ZFDSS Z compiler/type-checker called *zc*, the output *z.code* of which is the *conceptual model*, although it is very similar in style to an intermediate object language produced by a traditional compiler. In contrast the TranZit Transformation Engine (TTE) does not make use of an intermediate language format since it does not really benefit the process of producing an executable representation due to the reasons explored in section 5.1.6. Instead, the *internal schema objectbase* described in section 4.3.1 provides all the necessary information for TranZit to achieve a transformation. The benefit claimed by ZFDSS for an intermediate language is the ability to *expand* and *uncompile* this conceptual model back into the original Z schema definitions, for the purposes of automating schema calculus operations.

The executable representation produced by ZFDSS is based on the Prolog language, whereas that produced by TranZit is based on LISP. ZFDSS includes a separate Prolog translator *zp*, which converts the *z.code* produced by *zc* into Prolog on an individual schema basis. An important observation is that, in contrast to the *TranZit Transformation Engine* (TTE), *zp* makes no attempt to determine whether a particular predicate in the Z specification does in fact have an executable representation. The computability aspects of producing an executable representation are therefore ignored by *zp*, which is justified by the view that “most people work with a normal subset of Z”. In contrast, the TTE makes explicit the view that *recognition* of some possibly non-computable clauses in Z specifications is an important element in the transformation process, and hence the development of the *TranZit Transformation Assistant* is a key component in supporting the eclectic approach proposed in resolving these problems.

In summary, whilst TranZit and ZFDSS are concerned with similar fields of research, the solutions and strategies employed differ in several important respects highlighted above. In particular the motivation for transformation is very different in the two systems: In ZFDSS, the transformation exists mainly to support the animation process for the purposes of assessing the *quality* of a program produced from the original specification, as part of the overall Ceilidh system. Whilst this is a highly relevant goal, it contrasts sharply with the aim of the TranZit transformation process which is to produce an executable representation for the purposes of computer-based validation of system requirements.

With this in mind, the remainder of this chapter examines two case studies, which are specifically designed to highlight the use of *TranZit* as an integrated component within a practical animation environment.

6.4 Case Study I: A Library System

The first case study examines a simple library lending system based on Diller (1990), that is intended to store the loan and return of books. In addition the system incorporates a simple query mechanism to determine which books are currently on loan. This problem is typical of the level presented to undergraduates following a few weeks exposure to Z and the animation environment.

Firstly, a typical Z specification is offered that has been developed to meet the *ad-hoc* requirements identified above. This is clearly not the only solution or the most efficient, but it is intended to illustrate the use of the toolset for the purposes of animation. This example is also published in a slightly different form in Siddiqi *et al.* (1998).

6.4.1 Z Specification Development

The first element of the specification is to define given types to represent peoples names and unique book titles respectively. In particular we make the tacit assumption that the library holds a single copy of each book.

[PERSON, BOOK]

In addition two reports are defined indicating the loan status of a particular book.

REPORT ::= BookIsOnLoan | BookIsNotOnLoan

The first element of the specification is the Library state schema. This records the fact that there are borrowers, and that they can borrow up to a maximum of three books each.

library borrowers : \mathbb{P} PERSON loans : PERSON \rightarrow \mathbb{P} BOOK <hr/> $\forall p : \text{PERSON} \mid p \in \text{dom loans} \bullet p \in \text{borrowers} \wedge \#(\text{loans}(p)) \leq 3$

Borrowers can borrow books provided they are registered with the system, and have less than three books currently on loan.

borrow Δ library b? : BOOK p? : PERSON <hr/> p? \in borrowers (p? \notin dom loans \vee (p? \in dom loans \wedge $\#(\text{loans}(p?)) < 3$) loans' = loans \oplus { p? \mapsto loans (p?) \cup { b? } }
--

A borrower may return a book provided it belongs to the library, and is on loan to that borrower.

return Δ library b? : BOOK p? : PERSON <hr/> p? \in dom loans b? \in loans (p?) loans' = loans \oplus { p? \mapsto loans (p?) \setminus { b? } }

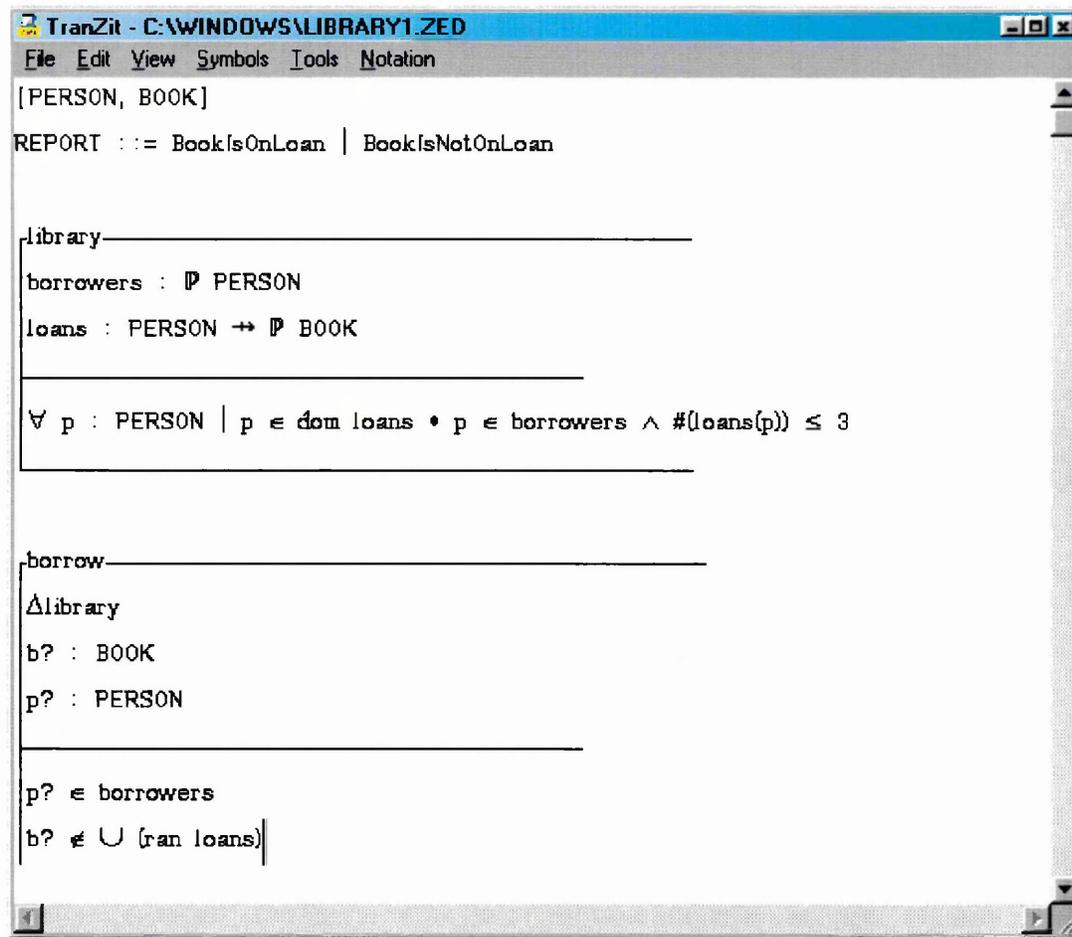


Figure 6-2: Capturing the Library Specification in TranZit

Using the *TranZit* analyser subsystem confirms that there are no syntax or type errors in this specification as shown in Figure 6-3, and it is possible to proceed to transformation.

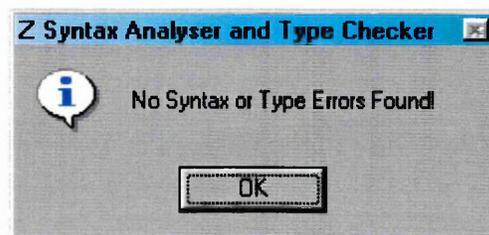


Figure 6-3: Screen Dump from TAS for Library Specification

The study proceeds by invoking the *TranZit* transformation engine. This produces the transformation system output screen as shown in Figure 6-4.

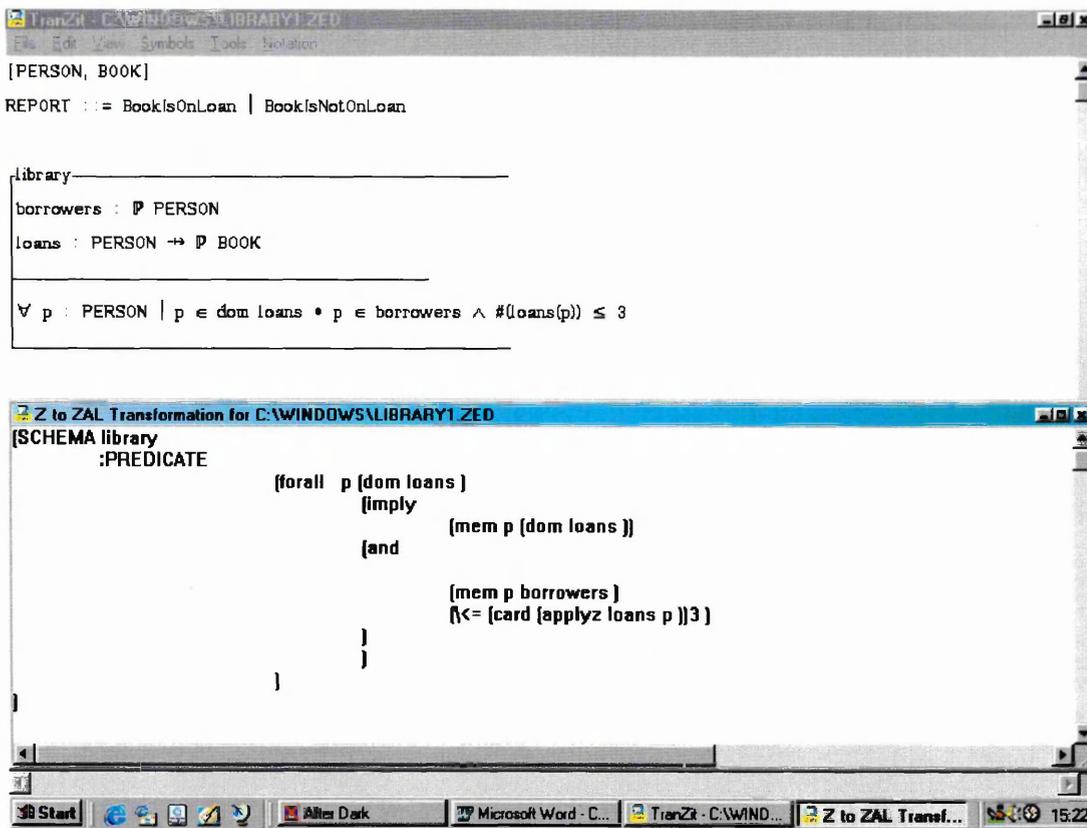


Figure 6-4: Screen Dump Showing TTE Output for Library Specification

For reference, the complete transformation of this specification as produced by *TranZit* is given below:

```
(SCHEMA library
  :PREDICATE
  (forall p (dom loans)
    (imply
      (mem p (dom loans))
      (and
        (mem p borrowers)
        (\<= (card (applyz loans p)) 3)
      )
    )
  )
)
```

```

(SCHEMA borrow
  :? ( b? p?)
  :INCLUDE delta_library
  :PREDICATE
  (and
    (mem p? borrowers )
    (not-mem b? (union-dis (ran loans )))
    (or
      (not-mem p? (dom loans ))
      (and
        (mem p? (dom loans ))
        (\< (card (applyz loans p? ))3 )
      )
    )
    (eqz 'loans' (override loans { #(p? (unionz
      (applyz loans p? ) {b? }))) )))
  )
)

```

```

(SCHEMA return
  :? ( b? p?)
  :INCLUDE delta_library
  :PREDICATE
  (and
    (mem p? (dom loans ))
    (mem b? (applyz loans p? ))
    (eqz 'loans' (override loans { #(p? (setsub
      (applyz loans p? ) {b? }))) )))
  )
)

```

```

(SCHEMA query
  :? b?
  :! rep!
  :INCLUDE psi_library
  :PREDICATE
  (or
    (and
      (exist p (dom loans )
        (and
          (mem p (dom loans ))
          (mem b? (applyz loans p ))
        )
      )
      (eqz rep! 'BookIsOnLoan )
    )
    (and
      (not
        (exist p (dom loans )
          (and
            (mem p (dom loans ))
            (mem b? (applyz loans p ))
          )
        )
      )
      (eqz rep! 'BookIsNotOnLoan )
    )
  )
)

```

```

(SCHEMA delta_library
  :INCLUDE (library library' )
  :PREDICATE
  t
)
(SCHEMA psi_library
  :INCLUDE (library library' )
  :PREDICATE
  (and
    ( eqz loans loans' )
    ( eqz borrowers borrowers' )
  )
)
(SCHEMA library'
  :PREDICATE
  (execute library (schema-rename library ( loans loans' )
    ( borrowers borrowers' ) ) )
)

```

Note that schemas *delta_library*, *psi_library* and *library'* have been generated by *the TranZit implicit schema resolution system*, since these are implicitly defined by the original Z specification.

It should also be noted that *TranZit* is able to perform the transformation of this specification with *no assistance from the user*. This is because, even though the specification contains potentially non-computable existential quantifications, *TranZit* is able to determine a suitable *enumeration function* for each transformed construct. It is now possible to proceed to the ZAL environment to interactively investigate the validity of each operation.

6.4.3 Animation in the ZAL Environment

It is important to realise that the power of the Windows environment allows us to have both the *TranZit* and ZAL animation environments active at the same time. Thus it is possible to refer either to the original specification or the animation environment quickly, and be able to adjust the specification as necessary as animation proceeds. With this system, it is easy to re-transform any modifications made to the original Z specification in *TranZit*, and re-load the result directly into the ZAL animation environment.

The ZAL animation runs in the Allegro LISP environment. Once ZAL is invoked the corresponding transformation produced by *TranZit* can be loaded, and the ZAL execution tool is invoked as shown in Figure 6-5:

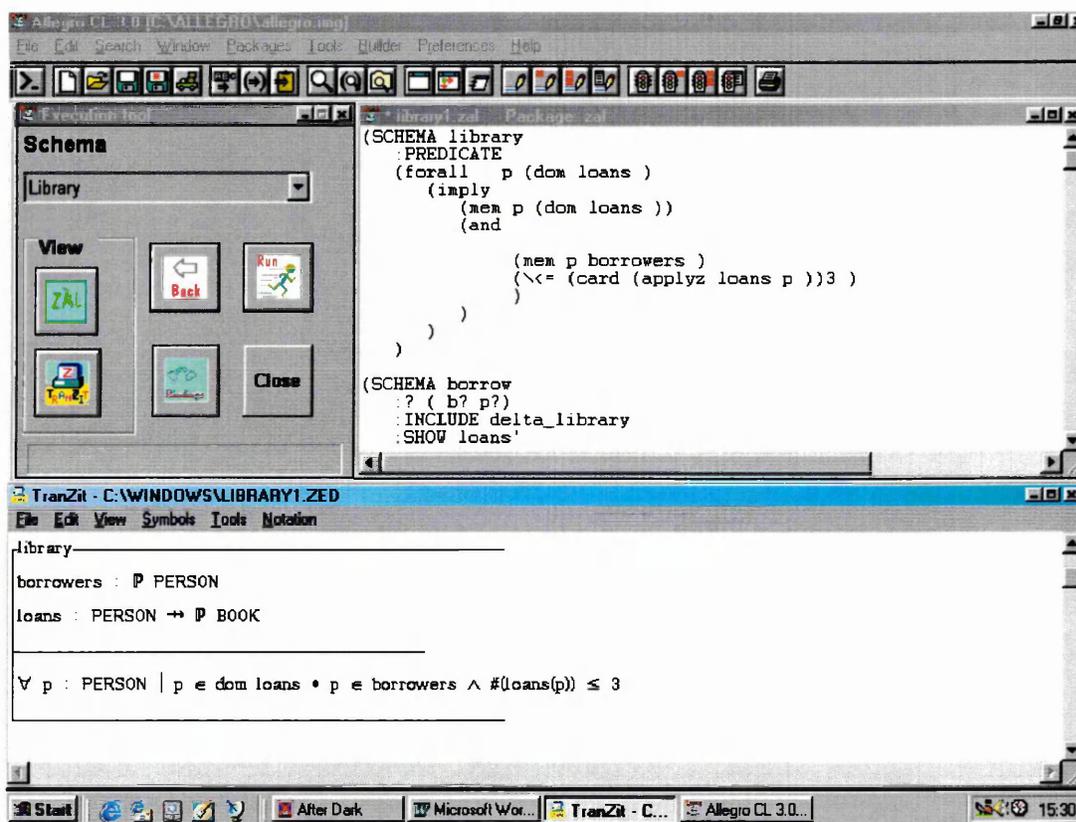


Figure 6-5: Using TranZit and ZAL to Animate the Library Specification

If desired, the ZAL button on the execution tool opens an edit window that contains the executable version of the specification as produced by *TranZit*. In addition the corresponding *Z specification* can be viewed in the *TranZit* window at the same time, as shown in Figure 6-5.

6.4.4 Creating Candidate Data

Before animation can begin, it is necessary to create some candidate data in order to populate the global data instances for the particular scenario we are interested in animating.

Firstly, we shall create a set of *borrowers* as:

```
Borrowers = {'ANNE 'BOB 'GRAHAM 'TOM 'ZOE}
```

The current state of the function *loans* (representing who has which books on loan) is then created as:

```
Loans = [#('ANNE {'MOLL_FLANDERS})  
         #('BOB {'TREASURE_ISLAND 'VANITY_FAIR})]
```

This is achieved through the binding browser in ZAL as shown in Figure 6-6:

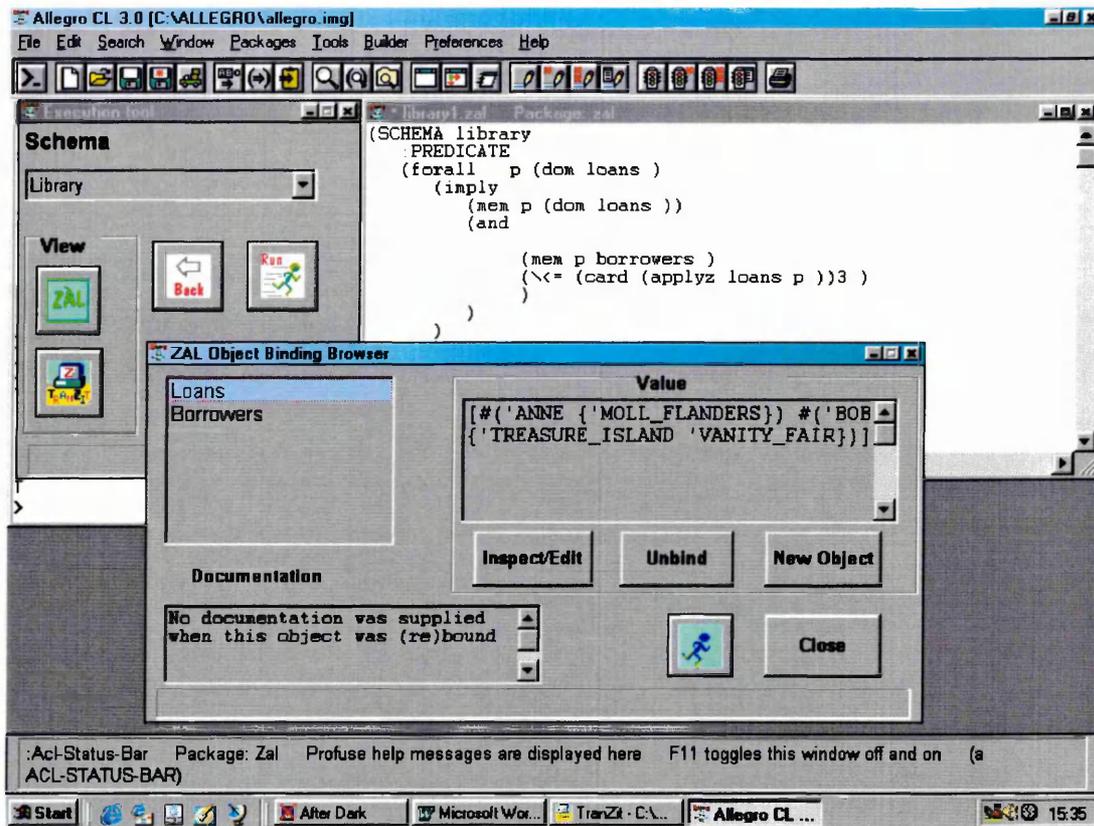


Figure 6-6: Creating Candidate Data for the Library Animation

6.4.5 Animating the Library Specification

It is now possible to begin to explore properties of the Library specification by animation of scenarios. To begin, an attempt is made to borrow a book from the library by executing the *borrow* schema. Selecting the schema from the execution tool and clicking the *Run* button in ZAL causes the system to prompt for values of the input variables *b?* and *p?*, as shown in Figure 6-7.

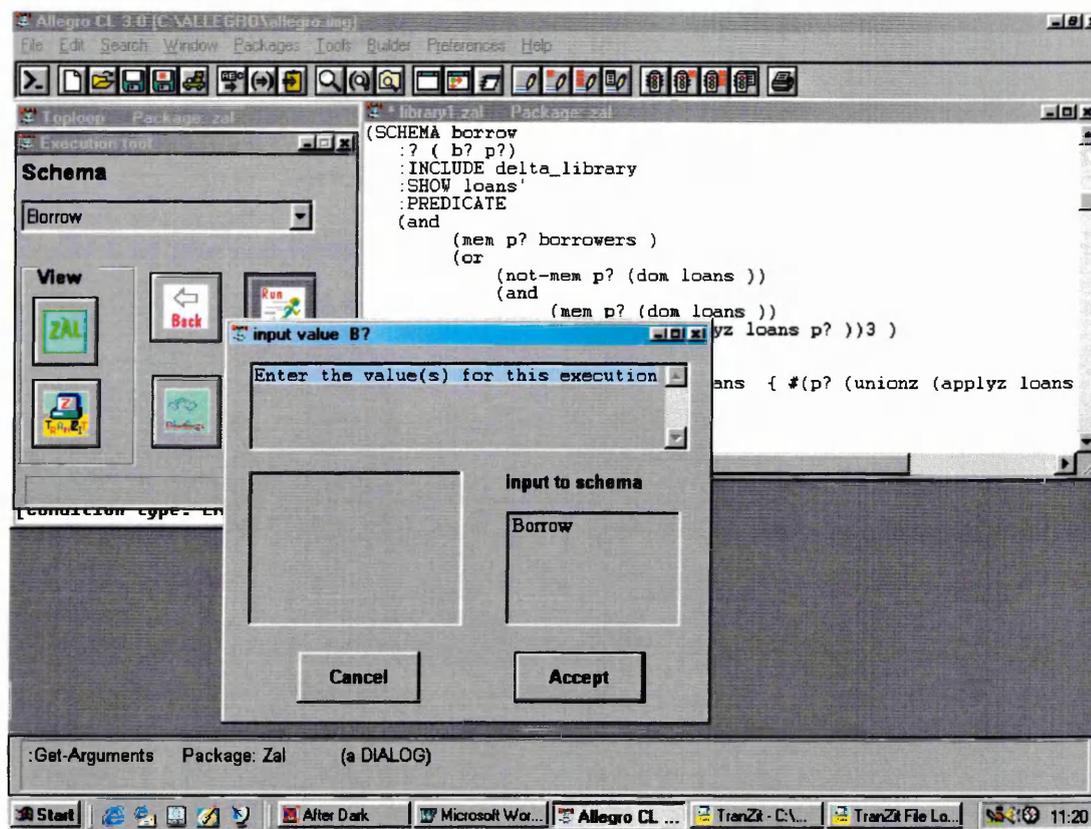


Figure 6-7: Executing the Library Animation

In this case we submit the value *Heidi* for *b?*. ZAL then prompts for a borrower *p?* and we submit *Anne*. ZAL then proceeds to execute the *borrow* schema and returns the value *TRUE*, to indicate success. The result of executing this schema can then be viewed in the ZAL *execution feedback* window, which in this case shows both the pre and post condition of the *loans* variable, as illustrated in Figure 6-8.

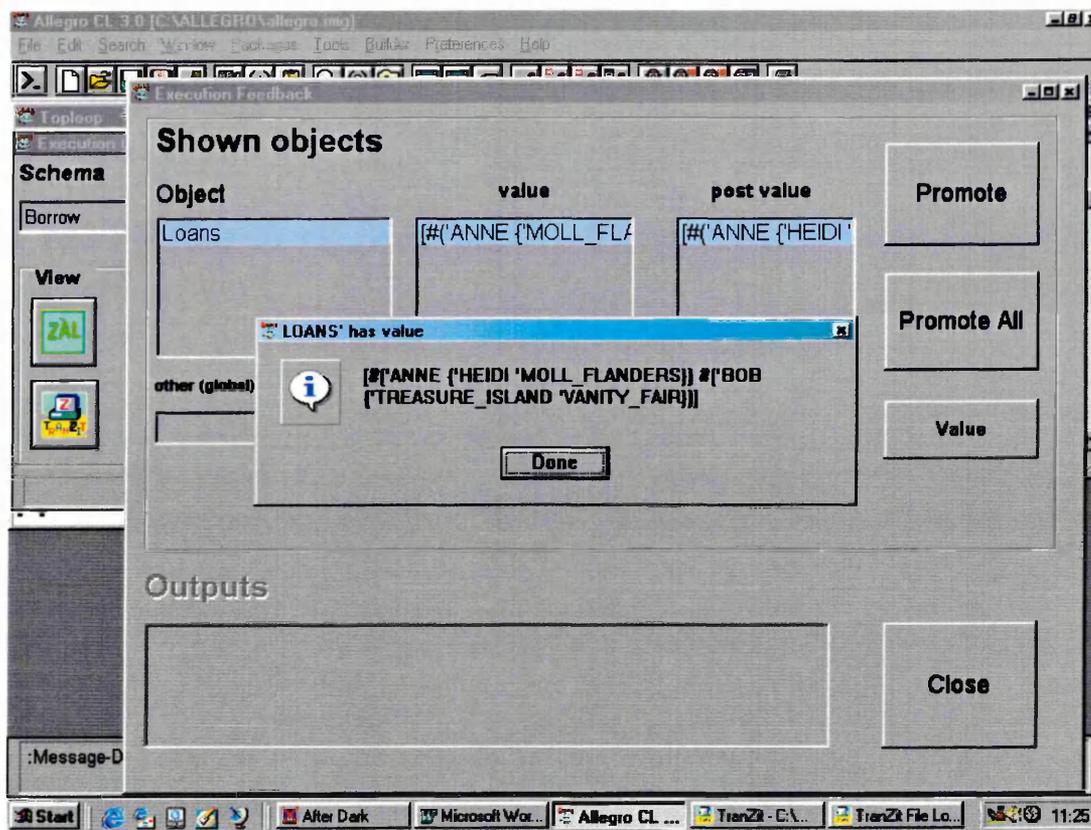


Figure 6-8: ZAL Execution Feedback Window on Executing Schema Borrow

In this case we see that the binding for *Anne* in *loans* has indeed been updated as expected. Depending upon the particular validation strategy, the user now has the option in ZAL to *promote* the post condition of *loans* to become the new current value. This allows a series of animation scenarios to be chained together.

The system can now be used to explore various additional properties of this specification. As a provocative example, it is decided to investigate what happens when an attempt is made to borrow a book that is already on loan (assuming the library holds a single copy of each book). In this case schema *borrow* is executed again, but this time the values of *Heidi* and *Bob* are supplied for *b?* and *p?* respectively, recalling that *Heidi* is already on loan to *Anne*. Surprisingly, executing schema *borrow* again returns the result *TRUE*, and the state of *loans'* shows that both *Anne* and *Bob* appear to have borrowed the same book, as seen in Figure 6-9.

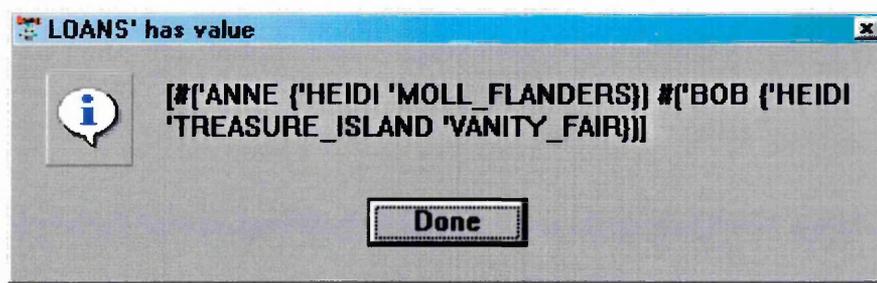


Figure 6-9: Provocative Testing of the Library Animation

It is unlikely that this behaviour should be allowed, hence it is clear that there is a need to assign a further predicate to the *borrow* schema in the original specification. A brief analysis, shows that the addition of the predicate:

$$b? \notin \cup (\text{ran loans})$$

will resolve this problem.

The original specification is modified accordingly in the *TranZit* window, and the *TranZit* transformation engine is invoked to produce a new executable representation. This is then re-loaded into ZAL as shown in Figure 6-10. Again, this transformation process is fully automated by *TranZit*.

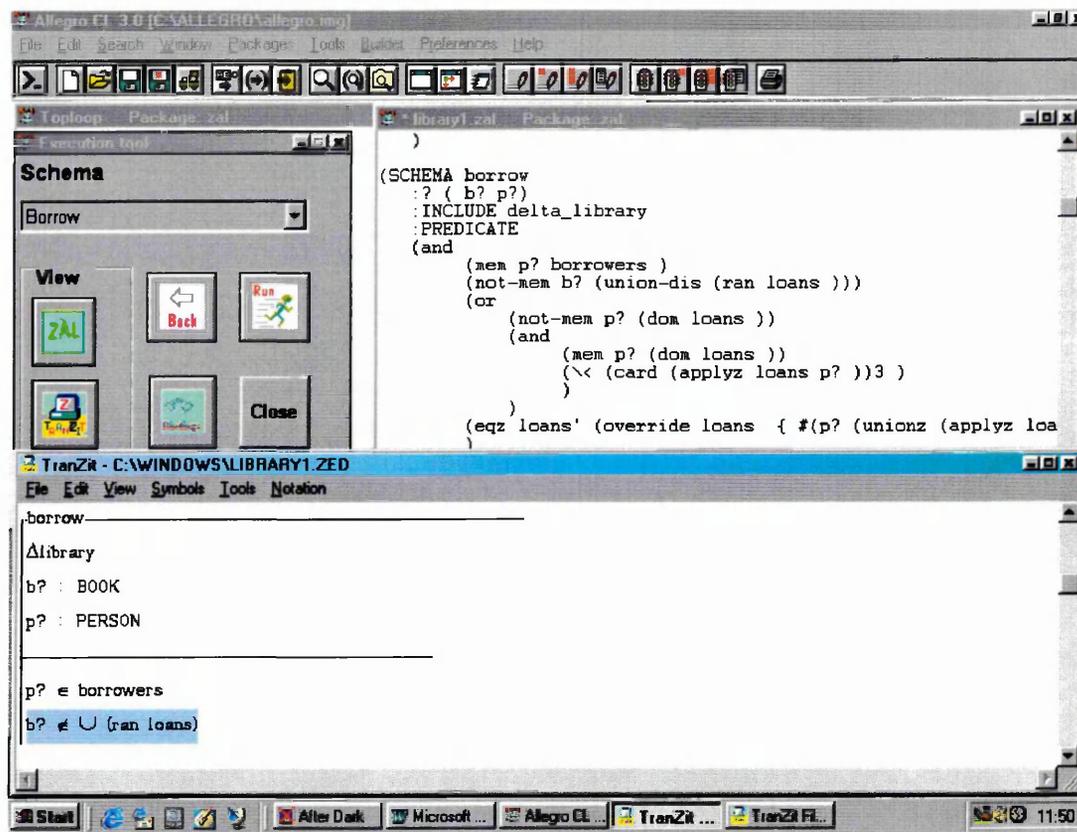


Figure 6-10: Executing the Revised Library Animation

The schema *borrow* is now executed again, supplying the same parameters as previously. This time, as expected, executing schema *borrow* returns *FALSE* as shown in Figure 6-11.

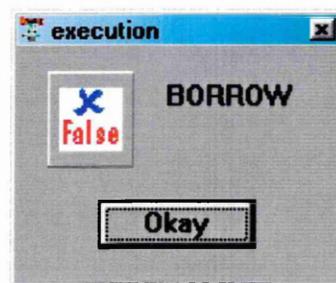


Figure 6-11: Output from ZAL on Executing the Revised Library Animation

6.4.6 Discussion

This case study indicates how it is possible to investigate the validity of requirements represented by a formal Z specification, using the *TranZit* and ZAL toolset. The approach taken by the toolset offers not only the possibility of validation, but also the important capability of *exploration* of the specification. Because the transformation process is automated as far as is possible by *TranZit*, it is possible to make exploratory changes to the original Z specification and rapidly investigate the effect of these changes in the animation environment. Thus, not only is it possible to confirm expected properties of the specification, it is also possible to explore provocative scenarios in order to uncover undesirable behaviour. A case in point in this example is the fact that the original specification assumes there to be more than one copy of each book. Hence the animation process helps to reveal what is *implicit* as well as *explicit* in a specification.

This approach also offers learning benefits. As reported in Siddiqi *et al.* (1998), it has been found that novice specification writers are often unsure whether they have added sufficient constraints within the schema predicates, to ensure the proper context for an operation. However, many users are experienced designers and programmers, who naturally make the association between *validation* of specifications and *testing* of programs. With the *TranZit* and ZAL toolset, it is possible for such people to bring many of the skills learned in testing programs, into the animation environment for the purposes of validation. Thus the toolset creates an important link between the unfamiliar validation domain and the familiar testing domain. This in turn brings a greater knowledge of individual Z constructs and also a deeper understanding of the wider context of writing formal specifications. Whilst the design of *TranZit* in particular, has gone to some lengths to preserve the divide between the formal specification and its executable representation, the fact that people view executable specifications as programs brings a number of benefits to the process as a whole.

6.5 Case Study II: The Telephone Network

As a more complex case study, Morgan's (1993) Telephone Network specification will be investigated. In this specification, connections may be established between pairs of telephones. If a request cannot be satisfied, because the called party is involved in another call (i.e. engaged), it will be stored for completion at some later time.

The original version of this specification published by Morgan (1987) contains an error, which is identified by *TranZit* as a type error. However, in what follows the discussion is confined to the amended version indicated above.

6.5.1 Z Specification Development

The specification itself is captured directly in *TranZit*, verbatim from Morgan's original paper. The specification is shown in its entirety below with natural language comments added for readability. These are delineated by the `/*` and `*/` sequences:

```
/*      Morgan's Telephone Network
      Firstly, we define a given set of telephone numbers. */
```

```
[ PHONE ]
```

```
/*      We proceed to represent a connection as a set of such numbers (this allows for
the possibility of multi-party call features, e.g. conference). */
```

```
CONN == PHONE
```

```
/*In a similar way, we proceed to represent call requests that have not yet terminated. */
```

```
reqs == CONN
```

```
/* There are two system invariants to be satisfied :
```

- only requested connections are active, and
- no phone may be engaged in more than one connection at any particular time.

```
The invariant is represented by Schema TN as follows: */
```

```
TN_____
|
| reqs, conns : CONN
|_____
|
| conns ⊆ reqs
| disjoint conns
|_____
```

```
/* Where  $disjoint\ S \leftrightarrow (c1, c2 : S \bullet c1 \neq c2 \Rightarrow c1 \cap c2 = \{\})$ 
```

However, an efficient TN would ensure that at any time, as many connections as possible are activated. That is the set of connections *conns* is maximal with respect to *TN*. This is represented by the corresponding *efficientTN* schema shown below: */

efficientTN

TN

$\neg(\exists \text{ conns0} : \text{CONN} \bullet$
 $\text{conns} \subset \text{conns0} \wedge$
 $\text{conns0} \subseteq \text{reqs} \wedge$
 $\text{disjoint conns0})$

/* Each of the network operations is described in terms of the state before, represented by schema efficientTN, and the state after, represented by schema efficientTN', and the phone from which it is initiated

 ph : PHONE

We collect these conditions in schema ΔTN , and impose the additional minimality constraint that a connection will never terminate unless termination is necessary to preserve the invariant. */

ΔTN

efficientTN
efficientTN'
ph? : PHONE

$\neg(\exists \text{ conns1} : \text{CONN} \bullet$
 $(\text{conns} \setminus \text{conns1}) \subset (\text{conns} \setminus \text{conns}') \wedge$
 $\text{efficientTN}' [\text{conns1} / \text{conns}'])$

/* We can now proceed to define operations on the abstract state space. The first operation *Call*, requests a connection between the initiating phone *ph* and the phone *dialled*. The request $\{ph, dialled\}$ is added to the set of requests, and the maximality constraint of efficientTN' ensures that if the request can be satisfied immediately, it will be. Similarly, the minimality constraint of ΔTN ensures no other changes will occur in *conns*. */

Call

Δ TN

dialed? : PHONE

$reqs' = reqs \cup \{ \{ph?, dialed?\} \}$

/ The Hangup operation terminates any connection in which the initiating phone ph is participating. Any such connection c is removed from the set of requests, which therefore also forces it to be removed from the set of connections. */*

HangUp

Δ TN

$reqs' = reqs \setminus \{c : conns \mid ph? \in c\}$

6.5.2 Capturing in TranZit and Transformation into ZAL

This specification is easily captured in *TranZit* in a very short period of time, and any errors eliminated using the TAS. The study then proceeds to the transformation phase. On running the transformation engine, *TranZit* presents the user with three requests from the *TranZit transformation assistant* as shown in Figure 6-12:

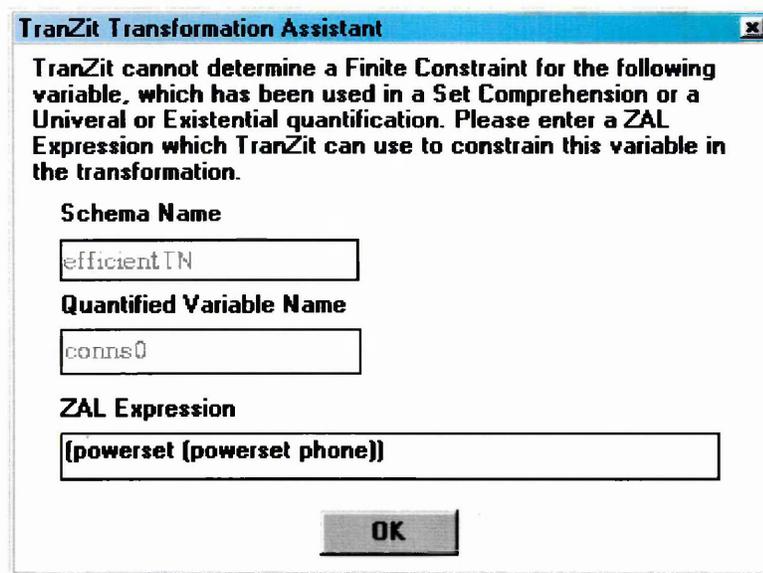


Figure 6-12: TranZit Transformation Assistant Request Dialog for Conns0

The transformation assistant is invoked by *TranZit*, since the existentially quantified variable *conns0* in schema *efficientTN* is unconstrained, being a search of the potential infinite set of sets, $P\text{ CONN}$. One could argue that in this case there is the potential to identify an *implicit* enumeration function, since CONN is defined as $P\text{ PHONE}$, and PHONE must be enumerated in order for the animation to make sense. However, since there is a level of indirection, it could be the case that the animator does not wish to enumerate the set PHONE , and prefers to work at the level of the set CONN . *TranZit* cannot make this judgement since it is dependent upon the mechanism that the animator wishes to adopt in his scenarios. The TranZit Transformation assistant therefore offers a choice. In this case, it is decided to enumerate the set PHONE , and therefore the corresponding enumeration function supplied for *conns0* is *(powerset (powerset phone))*, where *phone* is the user-supplied ZAL binding representing the set PHONE . This enumeration function is therefore equivalent to $P(P\text{ PHONE})$.

A similar issue arises with the existential quantifier for quantified variable *conns1* in schema ΔTN , as shown in Figure 6-13.

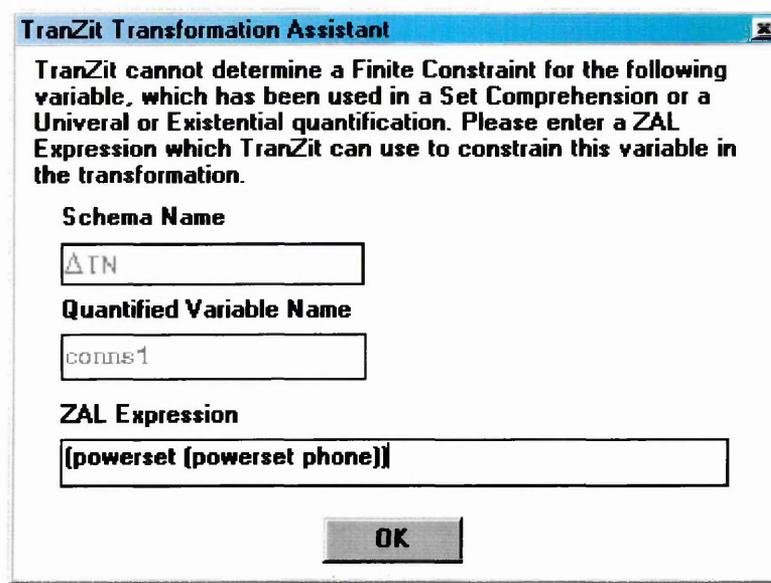


Figure 6-13: TranZit Transformation Assistant Request Dialog for Conns1

Again it is decided to supply the enumeration function (*powerset (powerset phone)*), representing $P(P \text{ PHONE})$, for the quantified variable *conns1*.

Finally, the Transformation Assistant presents the following request associated with the variable *c* used in the set comprehension in Schema *Hangup*, as shown in Figure 6-14.

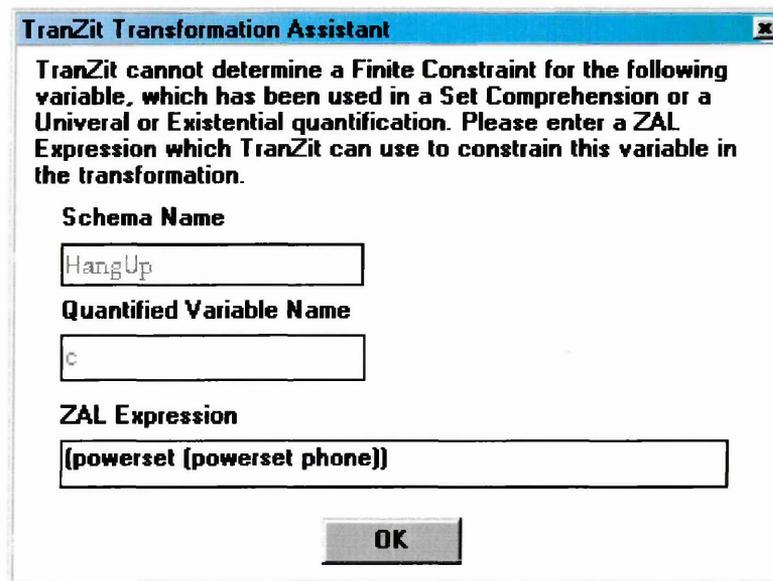


Figure 6-14: TranZit Transformation Assistant Request Dialog for Variable C

Again, working back through the schema hierarchy, it is found that *c* is actually of type P CONN, and hence the same argument applies as previously. Again it is decided to supply the enumeration function (*powerset (powerset phone)*).

Having supplied the required enumeration functions, the TranZit transformation engine automatically completes to produce the following executable representation in ZAL.

```
(SCHEMA TN
  :PREDICATE
  (and
    (subset conns reqs )
    (disjoint-dis conns )
  )
)

(SCHEMA efficientTN
  :INCLUDE TN
  :PREDICATE
  (not
    (exist conns0 (powerset (powerset phone))
      (and
        (true)
        (and
          (psubset conns conns0 )
          (subset conns0 reqs )
          (disjoint-dis conns0 )
        )
      )
    ))
)

(SCHEMA delta_TN
  :? ph?
  :INCLUDE ( efficientTN efficientTN' )
  :PREDICATE
  (not
    (exist conns1 (powerset (powerset phone))
      (and
        (true)
        (and
          (psubset (setsub conns conns1 )
            (setsub conns conns' ))
          (execute efficientTN'
            (schema-rename efficientTN' conns1 conns' ))
          )
        )
      )
    )
)
)
```

```

(SCHEMA Call
  :? dialled?
  :INCLUDE delta_TN
  :PREDICATE
  (eqz reqs' (unionz reqs { {ph? dialled? } })))
)

(SCHEMA HangUp
  :INCLUDE delta_TN
  :PREDICATE
  (eqz reqs' (setsub reqs (mksi 'c 'c (powerset (powerset phone))
    ' (mem ph? c )))))

(SCHEMA efficientTN'
  :PREDICATE
  (execute efficientTN
    (schema-rename efficientTN ( reqs reqs' )( conns conns' ) ) )
)

```

It is noted that in this case the specification writer has supplied an explicit schema ΔTN within the original specification, and hence the TranZit Transformation engine uses this definition to resolve associated references in preference to an internally generated implicit schema. However, reference is made to the implicitly defined schema *efficientTN'*, for which the Transformation engine provides a corresponding, automatically generated schema definition.

6.5.3 Animation in the ZAL Environment

The transformation shown above can now be loaded directly into the ZAL Animation environment as shown in Figure 6-15:

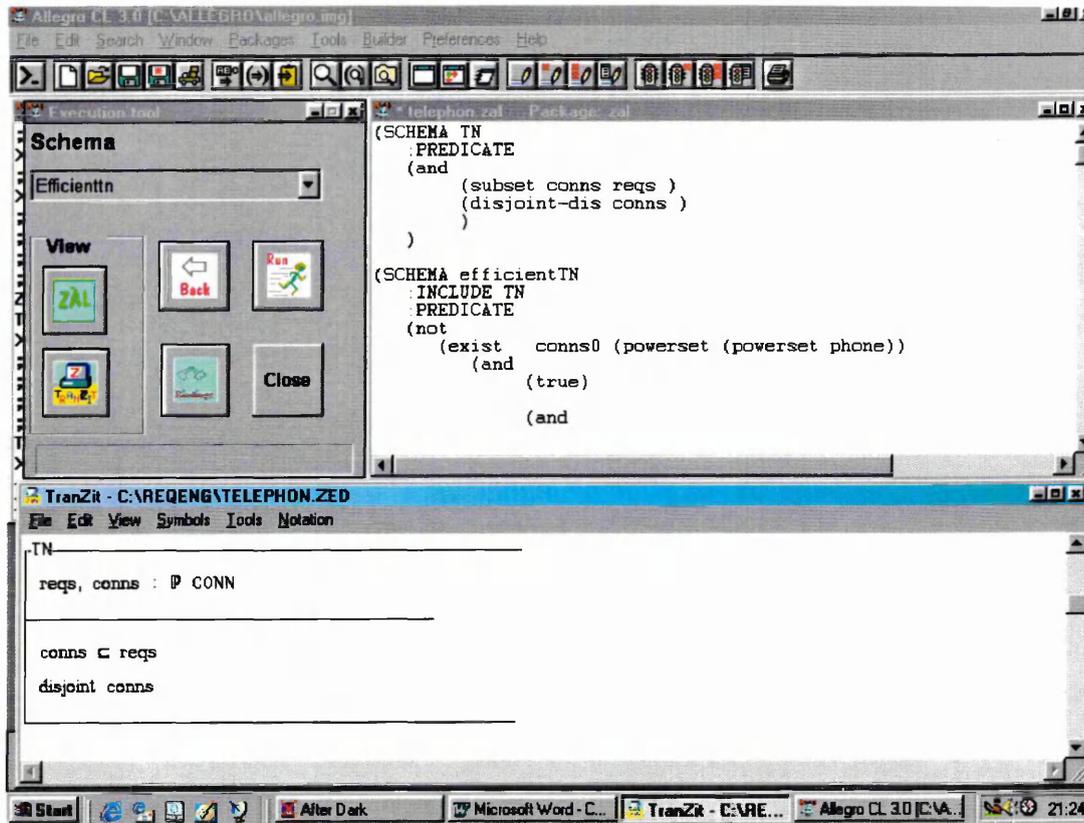


Figure 6-15: Executing the Telephone Network Animation

6.5.4 Creating Candidate Data

Before animation can begin, it is necessary to create some candidate data in order to populate the global data instances associated with the particular scenario to be investigated. The binding browser in ZAL is used for this purpose as described previously in the Library case study. This is illustrated in Figure 6-16.

Firstly, we shall create a simple set of *Phone* identities using the ZAL language as:

$$\text{phone} = \{ 1 \ 2 \ 3 \}$$

N.B. Being based on LISP, ZAL requires no commas to delineate individual elements of a set.

The current state of the sets *conns* and *reqs* is then established as:

- $\text{conns} = \{\{1 \ 2\}\}$ representing a connection between phones 1 and 2.
- $\text{reqs} = \{\{1 \ 2\} \{1 \ 3\}\}$ representing a connection request between phones 1 and 2 and phones 1 and 3.

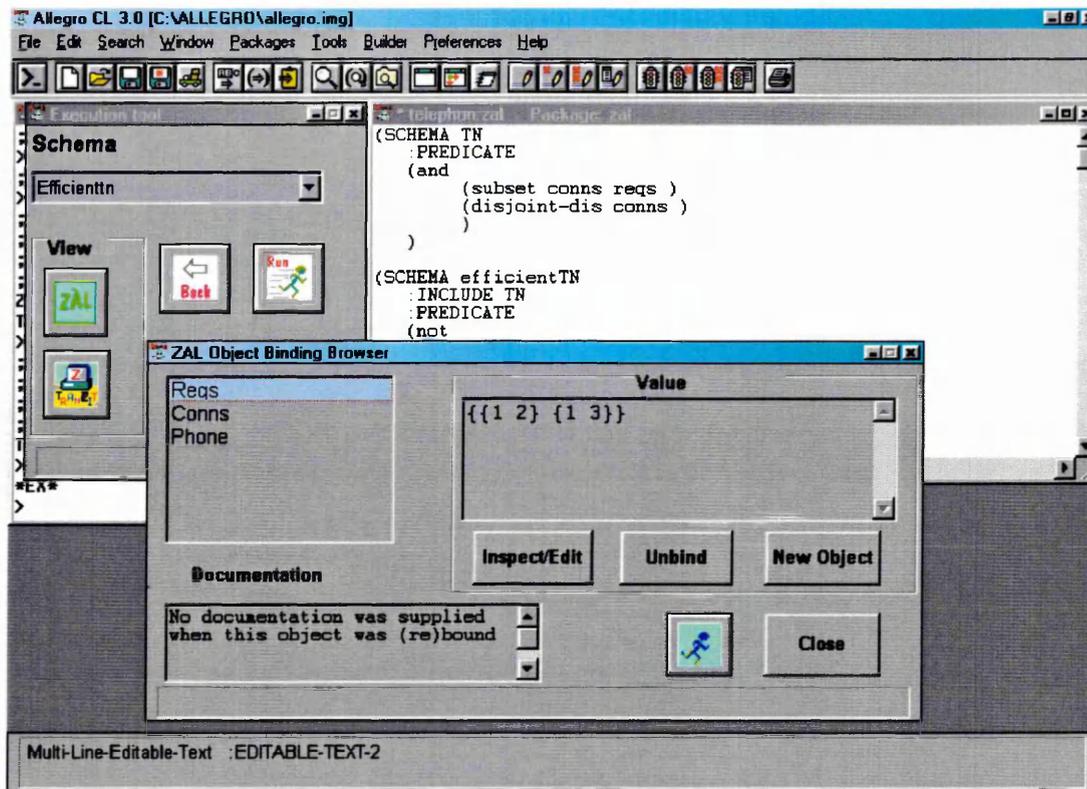


Figure 6-16: Creating Candidate Data for the Telephone Network Animation

6.5.5 Executing the Telephone Network Animation

It is now possible to explore properties of the specification by animation of scenarios. Firstly, executing Schema *TN* on this candidate data produces the result shown in Figure 6-17:

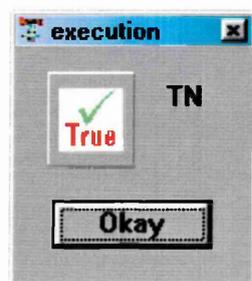


Figure 6-17: Result of Executing Schema TN

This is because both of the system constraints required by the network are satisfied for the candidate data set described previously. However, we now proceed to modify the candidate data as follows:

- $\text{conns} = \{\{2\ 3\}\}$ representing a connection between phones 2 and 3.
- $\text{reqs} = \{\{1\ 2\}\ \{1\ 3\}\}$ representing a connection request between phones 1 and 2 and phones 1 and 3.

In this case, executing schema *TN* returns the result *FALSE*, as shown in Figure 6-18:



Figure 6-18: Result of Executing Schema TN with Revised Candidate Data

This is because the current connection $\{2\ 3\}$ is not recorded as requested.

Similarly, we proceed to modify the candidate data again as:

- $\text{conns} = \{\{1\ 2\}\ \{1\ 3\}\}$ representing a connection between phones 1 and 2 and phones 1 and 3.
- $\text{Reqs} = \{\{1\ 2\}\ \{1\ 3\}\}$ representing a connection request between phones 1 and 2 and phones 1 and 3.

In this case, executing Schema *TN* again returns the result *FALSE*, as shown in Figure 6-19:

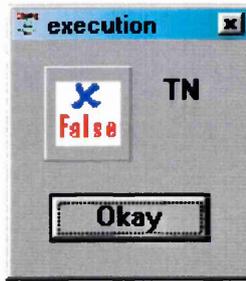


Figure 6-19: Result of Executing Schema *TN* with further Candidate Data Changes

This is because phone 1 cannot be involved in two connections at once.

Having understood the operation of schema *TN*, it is possible to explore the properties of schema *efficientTN*, to determine in what way it behaves differently. For this purpose the candidate data is populated as follows:

- $conns = \{\{1\ 2\}\}$ representing a connection between phones 1 and 2.
- $reqs = \{\{1\ 2\}\ \{3\}\}$ representing a connection request between phones 1 and 2 and phone 3 to itself.

Executing Schema *TN* returns the result *TRUE*, as shown in Figure 6-20:

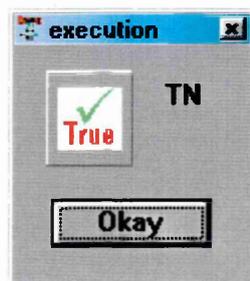


Figure 6-20: Result of Executing Schema *TN* as part of evaluating *EfficientTN*

However, executing Schema *efficientTN* for the same candidate data gives the result *FALSE*, as shown in Figure 6-21:

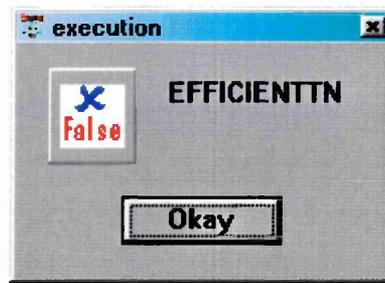


Figure 6-21: Result of Executing Schema EfficientTN

Executing schema *efficientTN* returns *FALSE*, because the candidate data for *conns* is not maximal. This is because there is nothing to prevent a connection between phone 3 and itself, however this is not recorded by the set *conns*. Thus we have demonstrated the property that *efficientTN* ensures that the set *conns* is maximal with respect to *TN*.

The study continues by exploring the properties of the *Call* operation. To achieve this the candidate data items are populated as follows:

- $conns = \{\{1\ 2\}\}$ representing a pre-condition connection between phones 1 and 2.
- $conns' = \{\{1\ 2\}\}$ representing a post condition connection between phones 1 and 2.
- $reqs = \{\{1\ 2\}\}$ representing the existing connection request between phones 1 and 2.

Executing the *call* schema, and supplying the parameters *dialled?* = 3 and *ph?* = 1, produces the result *TRUE*, and the execution feedback window appears as shown in Figure 6-22.

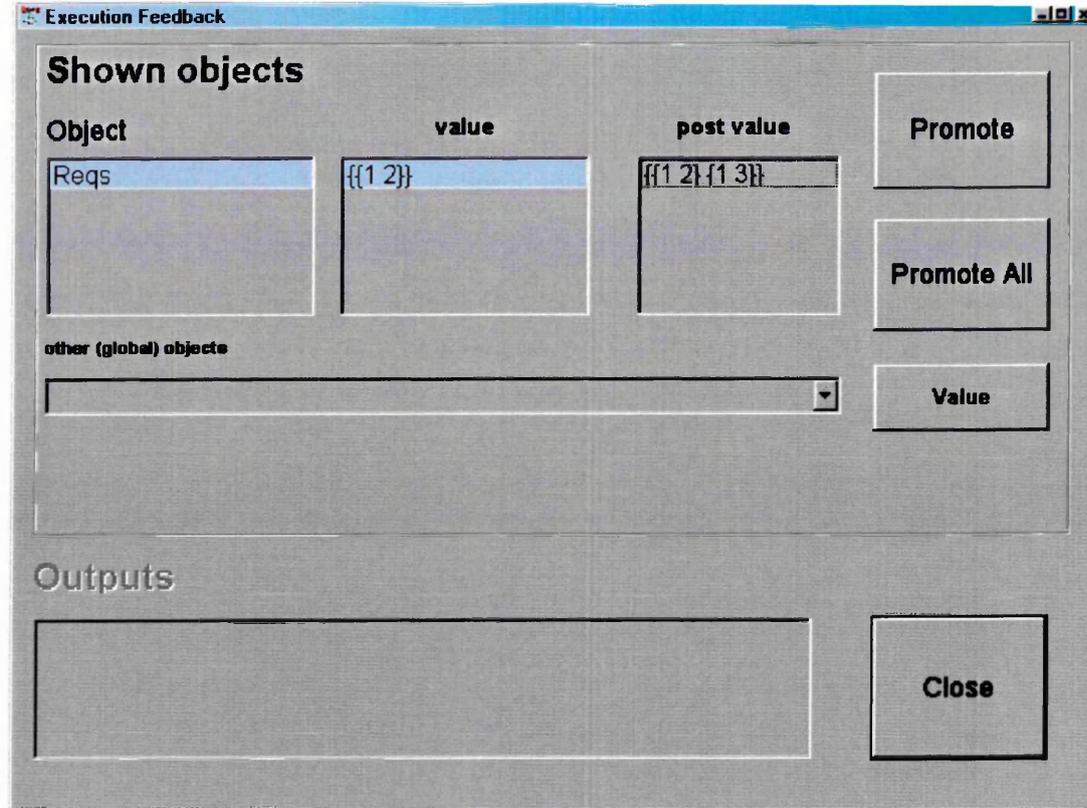


Figure 6-22: ZAL Execution Feedback Window on Executing Schema Call

The post value *reqs*' now identifies the fact that a call request between phones 1 and 3 has been made, but not yet connected.

Finally, the *hangup* operation is defined to terminate any connection in which phone *ph?* is involved. The study explores the expectation that hanging up a particular phone should automatically reconnect the phone to another if such a connection is in the set of requests *reqs*. That is, it is asserted that:

$$\exists \text{Hangup} \cdot (\text{conns}' \setminus \text{conns}) \neq \emptyset$$

To proceed, the candidate data is populated as follows:

- $\text{conns} = \{\{1\ 2\}\}$ representing a pre-condition connection between phones 1 and 2.
- $\text{conns}' = \{\{1\ 3\}\}$ representing a post condition connection between phones 1 and 3.
- $\text{reqs} = \{\{1\ 2\}\{1\ 3\}\}$ representing the existing connection requests between phones 1 and 2, and phones 1 and 3.

- reqs' = {{1 3}} representing the post condition connection request between phones 1 and 3, hypothesising the condition that when phone 1 hangs up breaking the existing connection with phone 2, it is immediately reconnected to phone 3.

The candidate data is created in ZAL using the browser interface described previously, as illustrated in Figure 6-23.

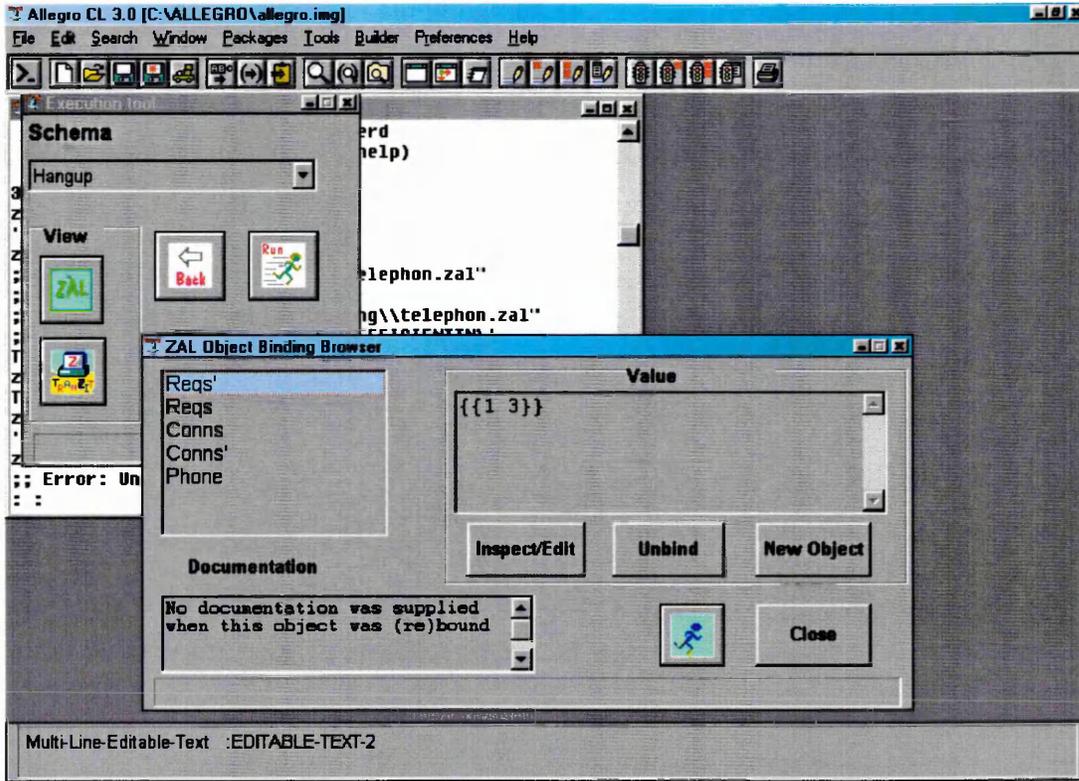


Figure 6-23: Creating Candidate Data for the Hangup Operation

On executing the *Hangup* schema, ZAL confirms the result *TRUE*, as shown in Figure 6-24.

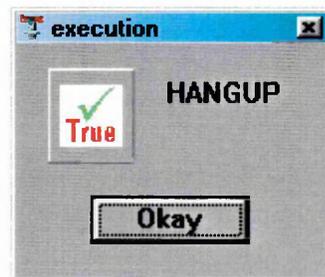


Figure 6-24: Result of Executing the Hangup Schema

This result actually mirrors the formal proof given in Morgan (1993) as follows:

$$\begin{aligned} & \text{Hangup} \wedge \\ & \text{reqs} = \{\{a,b\},\{a,c\}\} \wedge \\ & \text{conns} = \{\{a,b\}\} \wedge \\ & \text{ph?} = a \\ \Rightarrow & \\ & \text{reqs}' = \{\{a,c\}\} \wedge \text{conns}' = \{\{a,c\}\} \\ \Rightarrow & \\ & (\text{conns}' \setminus \text{conns}) \neq \emptyset \end{aligned}$$

6.5.6 Discussion

This case study demonstrates a different animation approach to that used in the previous library example. In the library case study, the user provided pre-conditions for state variables and used the ZAL environment to *calculate* post-conditions. This is because the original Z specification is written in an *operational* style, which lends itself to this approach. However, in the telephone network specification many of the state variable conditions are defined indirectly, and the imposition of a procedural mechanism in order to derive the value of an unknown is generally not feasible. The approach taken to animation is therefore one in which *hypotheses* are proposed which are confirmed (or not) by the user providing inputs and outputs together with pre and post conditions for state variables.

It could be argued that this specification lacks clarity, being written in an abstract mathematical style, which introduces fairly complex constraints on the values populating its state space. This is also revealed by the additional assistance required from the user in performing the transformation of this specification. The more abstract nature of the specification inherently utilises potentially non-computable Z constructs, which requires

the user to reason about the nature of the animation and provide corresponding constraints for *TranZit* to use in the associated executable representation.

Whilst the *Z* specification could be written in a more *procedural* style, there is nothing inherently wrong with the way in which it is currently captured and the transformation and animation tools must be able to deal with such problems if they are to be generally applicable. Hence this study shows how the *TranZit* transformation assistant effectively bridges the gap between the more abstract use of the *Z* notation, and the requirements for an executable representation.

6.6 Summary

This chapter has described the testing strategy adopted during system validation to ensure that the *TranZit* tool developed is a high quality software product, which is reliable, robust and fit for purpose. In addition, ensuring that the set of product requirements developed earlier in the research programme has been met by the current implementation in turn validates the tool itself. In addition to validating the product requirements, a user questionnaire has also been devised to validate that features of the tool have been implemented in the most *usable* way, and to provide important feedback on improvements and possible future development directions for *TranZit*.

The remainder of this chapter has described two case studies, which have been selected to demonstrate the operation of the *TranZit* editor, analysis subsystem and transformation engine working in conjunction with the *ZAL* animation environment. These specifications have been chosen on the basis of their contrasting styles, aiming to highlight the way in which the *TranZit* transformation engine deals with specifications written in an *operational* style and also specifications written in a more abstract mathematical style. The later case is also used to demonstrate the operation of the *TranZit* animation assistant and indicate likely conditions in which the assistant can resolve otherwise non-computable problems.

Together these case studies illustrate the use of *TranZit* as a key component in an integrated animation environment, providing the means whereby users can rapidly

capture and explore properties of Z specifications for the purposes of validation by execution.

The final chapter draws results and conclusions from this work by considering what has been achieved in this project, as well as exploring the possibilities for future research work. Lastly, general conclusions are drawn concerning the achievements of the overall project and the possibilities for formal specification and animation in the future.

7. Results and Conclusions

In this final chapter, the general results and conclusions of this research programme are identified. In particular, the achievements of the programme are reviewed against the original objectives in order to measure the overall success of the project. Secondly, the possibilities for future improvements and enhancement of the *TranZit* system are identified, in order to ensure that there is continuity of the work and a foundation for continued development. Finally, general conclusions are drawn concerning the overall project results and the future use of animation and formal methods in requirements engineering.

7.1 Review of Achievement Against Research Programme Objectives

The original objectives of the research programme were identified in section 1.3, and are reiterated below:

To research, implement and critically evaluate a CASE tool for requirements engineering, which will allow the capture of formal specifications and subsequently produce an executable representation of the specification, suitable for use as a rapid prototype within an animation system. The project should make a contribution to knowledge in terms of addressing the problems association with the transformation of non-executable specifications.

This was to be achieved by a number of objectives also outlined in section 1.3. It is now possible to establish how well the research programme has succeeded in achieving these objectives.

- *Objective 1: Definition of a process model for requirements engineering based on the use of the toolset.*

This research programme has identified and developed the REALiZE process, which forms the foundation upon which the integrated toolset consisting of *TranZit*, *ZAL* and *ViZ* is constructed. REALiZE embodies the concept of *validation of specifications by*

execution, which provides the important link between formal specification and specification validation by the customer. REALiZE is not a prescriptive process, but seeks to acknowledge that the task of requirements engineering involves human qualities such as creativity, elicitation, understanding and reasoning, whilst at the same time providing structure to the process in terms of the definition of tools to maximise the efficiency of specific tasks.

- *Objective 2: Definition and implementation of a computer-based tool that can be used to capture and store specifications efficiently in a formal notation.*

This research programme has focussed on the research and development of the *TranZit* tool. This tool has been successfully implemented a full-screen editor which makes use of the MS Windows platform as a basis for capturing formal specifications written in the Z notation. *TranZit* makes use of the standard Windows GUI to provide an integrated and efficient set of features for the construction and manipulation of Z specifications. This GUI has been extensively tested and trialled. In addition user questionnaires have shown that there is a general consensus that *TranZit* is well designed and presents information in a logical fashion to assist in the specification development process.

- *Objective 3: Definition and implementation of an analysis system for checking the internal consistency and correctness of the specification that is captured.*

The *TranZit* tool incorporates a highly optimised syntax and type checker subsystem termed the TranZit Analyser Subsystem (TAS). The TAS is able to identify a wide range of errors in a captured specification, based on Spivey's (1992) original Z notation grammar together with extensions from the developing Z base standard (Brien and Nicholls, 1992). The design of the TAS is based on innovative techniques combining traditional compiler technology with object-oriented data structures and Z grammar manipulation, to produce a highly efficient solution. User questionnaires have also identified that the TAS is considered to be one of the most useful features of the *TranZit* tool.

- *Objective 4: Definition and implementation of a computer-based mechanism to automate (as far as is possible) the transformation of the captured specification, into a procedural or executable representation, suitable for use as a rapid prototype in an animation system for the purposes of validating the captured specification by execution.*

The most important part of this research programme has been the research and development of the *TranZit Transformation Engine (TTE)*. The development of this engine has been based on extensive research into prototyping and animation techniques, coupled with an investigation into the non-computable aspects of formal specification languages. It has been demonstrated that the *TranZit Transformation Engine* is able to *automatically* transform a high proportion of Z notation constructs into a corresponding executable representation in the ZAL language. In addition, the TTE is supported by an innovative component termed the *computability analyser* which is able to determine a range of conditions under which automated transformation is possible, and elicit help from the user when these conditions do not apply. Using this novel, eclectic approach, *TranZit* is able to perform the transformation of a wide variety of Z notation constructs automatically and also provides a mechanism to resolve the transformation of Z notation constructs which are potentially non-computable.

- *Objective 5: Testing and Evaluation of what has been achieved including comparison with other computer-based requirements engineering tools, and demonstration of the efficacy of the solutions embodied through practical application in an animation environment.*

The *TranZit* tool has been validated by a carefully controlled development programme involving a series of testing phases conducted at strategic points in its development. This provides a high level of confidence in the quality of the delivered software product. In addition *TranZit* has been made available to a large number of staff and students at SHU and therefore exposed to a wide variety of specification problems. Very few problems have been reported with the tool itself and it is generally believed to be a reliable and stable product. In terms of achievement, *TranZit* has been successfully integrated with

the ZAL animation environment by the development of the transformation engine, and it has also been shown to be a sophisticated requirements engineering tool in its own right. These achievements have been evaluated by comparison with the *Formaliser* tool (Logica Inc., 1995) and the *ZFDSS* toolset (Zin, 1993), to highlight the unique characteristics presented by the *TranZit* tool in its approach to supporting computer-based Z specification construction and transformation to an executable representation. In addition, the practical application of the *TranZit* tool to a significant number of specification problems at SHU has demonstrated that the TranZit Transformation Engine is able to transform a wide variety of Z specifications directly into the ZAL language. These can then be imported directly into the ZAL animation environment to support the requirements engineering goal of *validation by execution*.

From the preceding discussion, it has been shown that the research and development of the *TranZit* tool has been successful in achieving all the original objectives of the research programme.

7.2 Opportunities for Further Research Work

Through the development and practical application of the *TranZit* tool, together with the results of user questionnaires, it has become apparent that there are several avenues of research that could be pursued in the future, based on this work.

The major area of future research should concentrate on enhancing the computability analyser of the TTE. Whilst this project has identified strategies and implemented solutions to address a number of key problems in the transformation of potentially non-computable clauses in Z, there is now the possibility to build on this work to increase the level of automation which can be achieved. As discussed previously in Chapter 5, enhanced strategies such as automated *re-writing* of clauses in the specification to make the transformation process easier, coupled with increased language intelligence in the recognition engine associated with identifying enumeration functions, suggest intriguing possibilities to further enhance the capabilities of the computability analyser. It is also likely that the ZAL language definition will mature to incorporate more elements of the Z

notation, allowing more complex specifications to be transformed by the TTE in the future.

Within the *TranZit* editor there is the possibility to include a number of useful teaching aids, which would enhance the process of specification construction. One of the major problems identified by most people new to the Z notation, is the understanding of the Z type system. There is therefore an opportunity to enhance the way in which *type* information is presented to the user. One of the suggested mechanisms would be to implement quick examination of types using *Windows tooltips*. Tooltips is becoming very popular in Windows programs, and allows instant feedback of information to the user. The user would simply place the cursor over an element of the specification or a selection, and tooltips would display the *type* of that variable or composite expression immediately. The aim is to afford learning as the specification is actually constructed, and to assist in identifying type errors in the specification earlier. A further extension to this idea would be to provide context-dependent help for syntax errors generated by the TAS, as this was identified in the user questionnaire as an area for potential improvement. A further teaching aid may also be the automatic expansion of schema expressions associated with the Z schema calculus, together with the automatic expansion of schemas which reference other schemas hidden by *inclusion*.

A second major enhancement to the *TranZit* editor would be to implement the Windows Multiple Document Interface (MDI), allowing several Z specifications to be loaded into the editor simultaneously. Whilst this is not required for smaller problems, a large requirements engineering task may require that many specifications are constructed, each pertaining to a particular sub-component of the system. The ability to be able to load and manipulate these specifications within the same editor session then becomes important, allowing the specifier to quickly refer to other parts of the system specification. In addition this approach could be supported by a *specification librarian*, in which *specification projects* are stored, each project consisting of many individual specifications in different states. The *specification project* would then be loaded into *TranZit*, making all the subordinate specifications accessible to the user by simple selection. The *specification librarian* may also support *configuration management* by maintaining

different versions of a particular specification in a controlled manner. This may be especially important in larger projects, where several people may be working on the same set of specifications.

In addition, the results of the user questionnaire suggest that the editor could be improved in the presentation and access of Z notation characters. Most people who criticised this feature of the editor identified the fact that access to characters via the menu system is too slow, and the *power-user option* of access via the Windows accelerator keys makes it too difficult to remember the required key combination. Clearly, what is required is an intermediate access method. One suggestion is the development of a *floating toolbar* on the menu system, which users could customise to add their particular set of commonly used Z notation characters. These characters would then be accessible via a single mouse click.

In terms of improving the interface between *TranZit* and the ZAL animation environment, one of the major enhancements which would increase the efficiency of information transfer would be to make use of the DDE server implemented in the Allegro LISP environment. At present, *TranZit* to ZAL information transfer uses an intermediate storage medium, which is either a separate file or the internal Windows *clipboard*. This enhancement would require the implementation of a DDE *Client* within *TranZit* to transfer information directly into the ZAL animation environment without intermediate clipboard or file storage.

In terms of specification portability, at present *TranZit* stores Z specification files in a proprietary format. To increase portability, it would be useful if *TranZit* could import and export Z specification files in the standard Z Interchange format (ZIF) defined in the Z base standard (Brien and Nicholls, 1992), as well as the popular L^AT_EX type-setting system (Lamport, 1985). In particular, the L^AT_EX format is supported by several other Z editor tools such as Spivey's (1988) *fuzz* package and the CADiZ tool from York Software Engineering (1991). More recently, the advent of the world-wide web has opened up possibilities to publish Z specifications on web pages, making them accessible over geographically wide areas. Some work on extending HTML to support Z has been

instigated at CERN, although no standards exists as yet. These portability enhancements would make the possibility for interchange of specifications between different tools more practicable, allowing integration with other toolsets addressing different elements of the requirements engineering task

In a wider context, the integration of the REALiZE toolset into other software engineering methods offers many avenues for research into the development of a methodology combining the capture, transformation and animation process with software design techniques. There is also the possibility of enhancing the REALiZE process by researching the requirements acquisition phase with a view to providing a front-end tool to assist in the formalisation of *informal* requirements at the beginning of the requirements engineering task. In addition, the extension of the REALiZE process to encompass *non-functional* requirements may provide a significant intellectual challenge in developing relationships between formal notations and system-oriented requirements such as performance and reliability factors.

7.3 General Conclusions

The approach to requirements engineering described in this thesis involves the application of executable formal specifications, based on the use of the Z notation, for the construction and validation of a rapid prototype of the system requirements. This approach seeks to maximise the strengths of formal specification, whilst at the same time minimising communication problems engendered by the mathematical knowledge required to understand the notation involved. This is achieved by a systematic process supported by computer-based tools, involving the capture of a clear, concise, precise and unambiguous specification of requirements in the *TranZit* tool, which can then be transformed to an executable representation in the ZAL language.

It has been argued that in using this approach, the behaviour and properties of the system embodied by the formal specification can be explored within a precise framework. This framework can be animated, and thus supports both the specification development itself and its subsequent validation by the user. The case studies described illustrate the potential value of formal specification and animation in improving comprehension and clarifying informal requirements.

Above all, the work presented herein has been guided by the principle of developing a *practical* tool, which combines the technical benefits offered by formal systems engineering techniques with the sociological and communication factors which influence the development of specifications for real-world applications. The understanding of the dynamics of specification development is critical in the design of a tool which adds value to this process, and the research undertaken to establish the underlying stakeholder interactions has been crucial in providing the foundation for the development of *TranZit*.

It is impossible to predict the eventual impact that the information revolution we are currently witnessing will have on the practical everyday aspects of our lives and the social fabric of our society in the future. However, what is clear is that the complexity of the software systems enabling this technology will continue to increase, and the efficient development of such systems will demand a corresponding increase in the power of software engineering methodologies, processes and tools.

Whilst software design techniques are well-developed and understood, effective techniques for requirements engineering continue to perplex all but the most mature of industrial development organisations. It is hoped that by the research and development of computer-based tools such as *TranZit*, industrial organisations will achieve their goal of developing quality system specifications, and the discipline of software engineering will advance to meet the challenges of the future.

Bibliography

- Abelson, H. and Sussman, G.J. (1985), *The Structure and Interpretation of Computer Programs*, McGraw-Hill. Maidenhead, UK.
- Agresti, W.W. (1986), *New Paradigms for Software Development*, IEEE Computer Society Press, Los Alamitos, California, USA.
- Alavi, M. (1984), "An Assessment of the prototyping approach to Information Systems Development," *Communications of the ACM*, 27, 6, 556 - 563.
- Alencar, A.J. and Goguen, J.A. (1991), "OOZE: An Object Oriented Z Environment," In *ECOOP '91 Proceedings*, P. America, Ed., Lecture Notes in Computer Science, 512, Springer-Verlag, Berlin, Germany, 180-199.
- Alford, M.W. and Lawson, J.T. (1979), *Software Requirements Engineering Methodology (Development)*, RADC-TR-79-168, U.S. Air Force Rome Air Development Centre, Griffiss AFB, NY, June 1979, (DDC-AD-A073132).
- Atkinson, W.D., Booth, J.P. and Quirk, W.J. (1991), "Model Action logic for the specification and validation of safety," In *Mathematical Structures for Software Engineering*, The Institute of Mathematics and its Applications Conference Series 27, Clarendon Press, Oxford, UK.
- Avison, D.E. and Fitzgerald, G. (1988), *Information Systems Development: Methodologies, Techniques and Tools*, Blackwell Scientific Publications, Oxford, UK.
- Avison, D.E. and Wood-Harper, A.T. (1991), "Information Systems Development Research: An exploration of ideas in practice," *The Computer Journal*, 2, 34, 98 - 112.
- Avizienis, A. and Wu, C-S. (1990), "A Comparative Assessment of Formal Specification techniques," In *Proceedings of the 5th Annual Knowledge-Based Software Assistant Conf*, 1990.
- B Core UK Limited. (1994), *B Toolkit Reference Manual*, Oxford Science Park, Oxford, UK.
- Backhouse, R.C. (1979), *Syntax of Programming Languages: Theory and Practice*, Prentice-Hall International, Englewood Cliffs, NJ, USA.

- Backus, J.W. (1959), "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference," In *Proceedings of International Conference on Information Processing*, UNESCO, 125-132.
- Balzar, R.M. (1985), "A 15 year perspective on Automatic Programming," *IEEE Transactions on Software Engineering*, 11, 11, 1257-1268.
- Barden, R., Stepney, S. and Cooper, D. (1994), *Z in Practice*, Prentice Hall, London, UK.
- Belkhouche, B. and Urban, J.E. (1986), "Direct implementation of Abstract Data types from Abstract Specifications," *IEEE Transactions on Software Engineering*, 12, 5, 649 – 661.
- Bergstra, J.A., Heering, J. and Klint, P. (1989), *Algebraic Specification*, ACM Press, New York, USA.
- Berzins, V., Luqi, K. and Yehudai, A. (1993), "Using Transformations in Specification-based Prototyping," *IEEE Transactions on Software Engineering*, 19, 5, 437 - 452.
- Bhabuta, L. (1989), "Balancing Systems and Organisational Needs: User Involvement in Requirements Analysis," In *Participation in Systems Development*, K. Knight, Ed., Kogan Page, 134 - 151.
- Bidoit, M. and Choppy, C. (1985), "Asspegique: An Integrated Environment for Algebraic Specifications," *Proceedings of TAPSOFT Conference*, 246-260.
- Boehm, B.W. (1981), *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, New Jersey, USA.
- Boehm, B.W. (1987), "Improving Software Productivity," *Computer*, September 1987, 43-57.
- Boehm, B.W. (1988), "A Spiral model of Software Development," *Tutorial: Software Engineering Project Management*, R.H. Thayer and M. Dorfman, Eds., IEEE Computer Society Press, Los Alamitos, California, 128 - 142.
- Boehm, B.W and In, H. (1996), "Identifying Quality-Requirement Conflicts," In *Proc. ICRE, 2nd International Conference on Requirement Engineering*, Colorado Springs, Colorado, IEEE Computer Society Press, Los Alamitos, California, USA, 218 - 219

- Booch, G. (1994). *Object oriented design with Applications*, Benjamin/Cummings, Redwood City, California, USA.
- Bowen, J.P. and Hinchey, M.G. (1995), "Seven more myths of Formal Methods," *IEEE Software*, 12, 4, 34 - 41.
- Bowen, J.P. and Gordon, M. (1994), "Z and HOL," In *Proceeding of the 8th Z User Workshop (ZUM'94)*, Cambridge, UK, J. Bowen and J. Hall, Eds., Workshops in Computing, Springer-Verlag, Berlin, 141-167.
- Bowen, J.P. and Gordon, M. (1995), "A Shallow embedding of Z in HOL," *Information and Software Technology*, 37, 5, 269-276.
- Brackett, L. (1990), "Case tools for Requirements Analysis and Software Design," In *Tutorial notes, 12th international conference on Software Engineering*, Nice, France.
- Breuer, P. and Bowen, J. (1994), "Towards Correct Executable Semantics for Z," In *Proceedings of the 8th Z Users Workshop (ZUM94)*, Cambridge, J. Bowen and J. Hall, Eds., Workshops in Computing, Springer-Verlag, Berlin, 185 – 212.
- Brien S.M. and Nicholls J.E. (1992), *Z Base Standard Version 1.0*, Oxford University Computing Laboratory Programming Research Group, Technical Monograph PRG-107, Oxford University Press, Oxford, UK.
- Brooks, F.P. Jr (1987), *Report of the Defense Science Board Task Force on Military Software*, September 1987, Office of the Under Secretary of Defense for Acquisition, U.S. Department of Defence, Washington D.C., USA.
- Brown, P.G. (1991), "QFD: Echoing the voice of the Customer," in *AT&T Technical Journal*, March-April 1991, 18 – 32.
- Bustard, D.W. (1994), "Progress Towards RACE, a 'Soft-Centred' Requirements Definition Method," In *Proceedings of the 1st IFIP/SQI International Conference on Software Quality and Productivity*, Hong Kong, Chapman & Hall, 29-36.
- Bustard, D.W. and Dobbin, T.J. (1996), "Integrating Soft Systems and Object-oriented Analysis," In *Proceedings of the 2nd International Conference on Requirements Engineering ICRC'96*, April 15th-18th, Colorado Springs, IEEE Computer Society Press, Los Alamitos, California, USA, 52 – 59.

- Cadre Technologies Inc. (1990), *Teamwork Environment Reference Manual*, Release 4.0, DX046XX4A, December 1990, Cadre Technologies Inc, USA.
- Carrington, D., Duke D., Duke, R., King, P., Rose, G.A. and Smith, G. (1990). "Object-Z: An object-oriented extension to Z," In *Formal Description Techniques II, (FORTE'98)*, North Holland, 281-296.
- Checkland, P.B. (1981), *Systems Thinking, Systems Practice*, John Wiley, Chichester, UK.
- Checkland, P.B. (1995), "Model Validation in Soft Systems Practice," *Systems Research, 1*, 12, 46 – 54.
- Cherniavsky, J.C. (1990), "Software Failures attract Congressional Attention," *Computer Research Review, 2*, 1, 4 - 5.
- Chomsky, N. (1956), "Three models for the Description of Language," *IEEE Transactions on Information Theory, IT2*, 113-124.
- Ciancarini, P., Cimato, S. and Mascolo, C. (1997), "Engineering Formal Requirements: An Analysis and Testing Method for Z documents," *Annals of Software Engineering, 3(1997)*, 189 – 219.
- Coad, P. and Yourdon, E. (1991), *Object Oriented Analysis*, Second Edition, Prentice Hall, Englewood Cliffs, New Jersey, USA.
- Collins, B.P., Nicholls, J.E. and Sørensen, I.H. (1988), *Introducing Formal Methods: The CICS Experience with Z*, Technical Report, Programming Research Group, Oxford University. Oxford, UK.
- Cusack, E. (1991), "Object-oriented Modelling in Z," In *ECOOP '91 Proceedings*, P. America, Ed., Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany.
- Cusack, E. and Wezeman C. (1993), "Deriving Tests for Objects specified in Z," In *Z User Meeting 1992*, J. Nicholls, Ed., Workshops in Computing, Springer-Verlag, Berlin.
- Damon, C. and Jackson, D. (1996), "Efficient Search as a means of Executing Specifications," In *Proceedings of TACAS'96*, T. Margaria and B. Steffens, Eds., Lecture Notes in Computer Science, 1055, Springer-Verlag, Berlin, 70-86.
- Daniels, A. and Yeates, D.A. (1971), *Basic Training in Systems Analysis*, Second Edition, Pitman Press, London, UK.

- Davis, A.M. (1988), "A Comparison of Techniques for the Specification of External System behaviour," *Communications of the ACM*, 31, 9, 1098-1115.
- Davis, A.M. (1993), *Software Requirements: Objects, Functions and States*, Prentice Hall International, Englewood Cliffs, New Jersey, USA.
- Davis, A.M., Overmyer, S., Jordan, K., Caruso, J., Dandashi, F., Dinh, A., Kincaid, G., Ledebor, G., Reynolds, P., Sitaram, P., Ta, A. and Theofanos, M. (1993), "Identifying and Measuring Quality in Software Requirements Specification," In *Proceedings of the 1st International Software Metrics Symposium*, 1993, 141-152.
- Davis G.B. (1981), "Information Analysis for Information Systems Development," In *Systems Analysis and Design: A Foundation for the 1980's*, W.W. Cottermann, J.D. Cougar, N.L.Enger and F. Harold, Eds.,
- DeMarco, T. (1978), *Structured Analysis and System Specification*, Yourdon Inc, New Jersey, USA.
- Dick, A.J., Krause, P.J. and Cozens, J. (1990), "Computer Aided Transformation of Z into Prolog," *Z User Workshop*, J.E. Nicholls, Ed., Workshops in Computing, Z User Group, Springer-Verlag, Berlin, 71 - 85.
- Diller, A. (1990), *Z: An Introduction to Formal Methods*, John Wiley and Sons, Chichester, UK.
- Doma, V. and Nicholl, R. (1991), "EZ: A System for Automatic Prototyping of Z Specifications," *VDM'91: Formal Software Development Methods*, S. Prehn and W.J. Toetenel, Eds., Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany.
- Dorfman, M. (1997), "Requirements Engineering," *Software Requirements Engineering 2nd Edition*, R.H Thayer and M. Dorfman, Eds., IEEE Computer Society Press, Los Alamitos, California, USA, 7 – 22.
- Dorfman, M. and Thayer, R.H. (1990), *IEEE standard 610-12: Standard, Guidelines and Examples on System and Software Requirements Engineering*, IEEE Computer Society Press, Los Alamitos, California, USA.
- Downs, E., Clare, P. and Coe, I. (1988), *Structured Systems Analysis and Design Methodology: Application and Context*, Prentice-Hall, Hemel Hempstead, Herts, UK.

- Durr, E., Duursma, A. and Plat, N. (1994), *VDM++ Language Reference Manual*, Technical Report AFRO/CG/ED/LRM, V9.1, May 1994, CAP Gemini Innovation, UK.
- Ebert, C. (1997), "Dealing with Nonfunctional Requirements," *Annals of Software Engineering*, 3(1997), 367 – 396.
- Eisenbach, S. (1987), *Functional Programming: Languages, Tools and Architectures*, Ellis Horwood, Chichester, UK.
- Elmstrøm, R., Larsen, P.G. and Lassen P.B. (1994), "The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specification," *ACM SIGPLAN*, 29, 9, 77 - 81.
- Floyd, C., Mehl, W., Reisen, F., Schmidt, G. and Wolf, G. (1989), "Out of Scandanavia: Alternative approaches to systems design and systems development". *Human Computer Interaction*, 4, 235 – 250.
- Flynn, D.J. (1992), *Information Systems Requirements: Determination and Analysis*, McGraw-Hill, Europe.
- Forberg K. and Mooz H (1997), "System Engineering Overview," *Software Requirements Engineering*, 2nd Edition, R.H. Thayer and M. Dorfman, Eds.,IEEE Computer Society Press, Los Alamitos, California, USA, 44 – 72.
- Fuchs, N.E. (1992), "Specifications are (preferably) executable," *IEEE Software Engineering Journal*, 7, 5, 323 - 334.
- Gane, C. and Sarson, T. (1979), *Structured Systems Analysis: Tools and Techniques*, Prentice Hall, New York, USA.
- George, C., Haff, P., Havelund, K., Haxthausen, E., Milne, R., Prehn, S., Wagner, K.R. and Nielson, C. (1992), *The RAISE Specification Language*, Prentice-Hall, New York, USA.
- George, C. and Prehn, S. (1992), *The RAISE Justification Handbook*, Technical Report, LA-COS/CRI/DOC/7/V4, Computer Resources International, October 1992.
- Gladden, G.R. (1982), "Stop the Life-Cycle, I want to get off," *ACM Software Engineering Notes*, SE-7, 2, 35 - 39.
- Glasson, B.C. (1984), "Guidelines for User Participation in the Systems Development Process," In *Human Computer Interaction (Interact '84)*, B. Shackel, Ed., North Holland.

- Goguen, J.A. and Meseguer, J. (1982), "Rapid Prototyping in the OBJ Specification Language," *ACM SIGSOFT Software Engineering Notes*, 7, 5, 75 - 84.
- Goguen, J.A. and Linde, C. (1993), "Techniques for Requirements Elicitation," In *Proceeding of the IEEE International Symposium on Requirements Engineering*, San Diego, California, IEEE Computer Society Press, Los Alamitos, California, USA, 152 - 164.
- Goguen, J.A. and Winkler, T. (1988), *Introducing OBJ3*, SRI International, USA.
- Goguen, J.A. and Wolfram, D. (1990), "On Types and FOOPS," In *Proceeding of the Working Conference on Database Semantics*, Windermere, Lake District, UK, July 1990.
- Goldberg, A. and Robson, D. (1983), *Smalltalk-80: The Language and its implementation*, Addison-Wesley, Massachusetts, USA.
- Goldschlager, L. and Lister, A. (1982), *Computer Science: A Modern Introduction*, Prentice-Hall International, Englewood Cliffs, New Jersey, USA.
- Gomaa, H. and Scott, D.B.H. (1981), "Prototyping as a tool in the specification of User Requirements," In *Proceeding of the 5th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, California, USA. 333 - 342.
- Gomma. H. (1997), "The Impact of Prototyping in Software System Engineering," *Software Requirements Engineering 2nd Edition*, R.H Thayer and M. Dorfman, Eds., IEEE Computer Society Press, Los Alamitos, California, USA, 431 - 440.
- Greenbaum, J. and Kyng, M. (1991), *Design at Work: Co-operative Design of Computer Systems*, Hillsdale, New Jersey, USA, Lawrence Erlbaum.
- Gries, D. (1971), *Compiler Construction for Digital Computers*, John Wiley and Sons, New York, USA.
- Gries, D. (1981), *The Science of Programming*, Springer-Verlag, New York, USA.
- Guttag, J.V. and Horning, J.J. (1993), *LARCH: Language and Tools for Formal Specification*, Springer-Verlag, New York, USA.
- Hall, A. (1990), "Seven Myths of Formal Methods," *IEEE Software*, 7, 5, 11 - 19.
- Harel, D. (1987), "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, 8, 231-374.

- Hartson H.R. and Smith E.C. (1991), "Rapid Prototyping in Human-Computer Interface Development," *Interacting with Computers*, 3, 1, 51 – 62.
- Hasselbring, W. (1994), "Animation of Object-Z specifications with a Set-Oriented Prototyping Language," In *Proceedings of the 8th Z User Workshop*, Cambridge, UK, J. Bowen and J.Hall, Eds., Workshops in Computing, Springer-Verlag, Berlin, Germany. 337 – 358.
- Haughton, H. and Lano, K. (1995), *B Abstract Machine Notation: A Reference Manual*, McGraw-Hill, London, UK.
- Hauser, J.R. and Clausing, D. (1988), "The House of Quality," *Harvard Business Review*, 66, 3, 63 – 73.
- Hayes, I.J. and Jones, C.B. (1989), "Specifications are not (necessarily) executable," *Software Engineering Journal*, November 1989, 330 - 338.
- Hayes, I.J. and Mahoney, B. (1992), "A Case Study in Timed Refinement: A Mine Pump," *IEEE Software*, 18, 9.
- Hayes, I. (1993), *Specification Case Studies: 2nd Edition*. Prentice Hall International, Hemel Hempstead, Hertfordshire, UK.
- Heckmatpour, S. (1988), *Lisp and Symbol Manipulation*, Open University Press, UK.
- Heckmatpour, S. and Ince, D. (1988), *Software Prototyping, Formal Methods and VDM*, Addison-Wesley, Wokingham, UK.
- Henderson P. and Minkowitz, C. (1985), "The me-too method of Software Design," University of Stirling, Department of Computing Science, FPN-10, Oct 1985.
- Henderson, P. (1986), "Functional Programming, Formal Specification and Rapid Prototyping," *IEEE Transactions on Software Engineering*, SE-12, 2.
- Hoare, C.A.R. (1969), "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, 12.
- Hoare, C.A.R. (1985), *Communicating Sequential Processes*, Prentice-Hall International, Englewood Cliffs, New Jersey, USA.
- Holbrook, H, III. (1990), "A Scenario-based methodology for conducting Requirements Elicitation," *ACM SIGSOFT Software Engineering Notes*, 15, 1, 95-104.
- Holub, A.I. (1990), *Compiler Design in C*, Prentice-Hall International, Englewood Cliffs, New Jersey, USA.

- Hörcher, H-M. (1994), "Animation and Prototyping of Implicit Specifications," *Geschäftsbereich*, DST Deutsche System-Technik GmbH, May 13, 1994.
- Hsai, P. and Yaung, A.T. (1988), "Screen-Based Scenario Generator: A Tool for Scenario-based Prototyping," In *Proceeding of the Hawaii International Conference on Systems Sciences*, Jan 4th-7th, Honalulu, USA, 455 – 461.
- Iachini, P.L. and Giovanni, R. Di. (1990), "HOOD and Z for the development of Complex Software Systems," *VDM and Z, VDM 90, 428, Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany. 262 – 289.
- IEEE (1984), "IEEE Guide to Software Requirements Specifications," IEEE Std 830-1984. IEEE inc, 345 East 47th Street, New York, NY 10017, USA.
- IEEE (1993), "IEEE Recommended Practise for Software Requirements Specifications," IEEE Std 830-1993. IEEE inc, 345 East 47th Street, New York, NY 10017, USA.
- Jackson, M.A. (1975), *Principles of Program Design*, Academic Press, London, UK.
- Jackson, M.A. (1983), *Systems Development*, Prentice-Hall, Englewood Cliffs, New Jersey, USA.
- Jackson, D. (1994), "Abstract Model Checking of Infinite Specifications," In *Proceedings of the 2nd International Symposium of Formal Methods Europe (FME)*, Barcelona, Spain, M. Naftalin, T. Denvir and M. Bertran, Eds., Lecture Notes in Computer Science, 873, Springer-Verlag, Berlin, Germany, 519-531.
- Jalote, P. (1987), "Synthesising Implementation of Abstract data types from Axiomatic Specification," *Software Practise and Experience*, 17, 11, 847-858.
- Jarke, M. and Pohl, K. (1994), "Requirements Engineering in 2001: (Virtually) managing a changing Reality," *Software Engineering Journal*, November 1994.
- Jones, C.B. (1990), *Systematic Software Development using VDM*, Prentice-Hall International , Englewood Cliffs, New Jersey, USA.
- Johnson, M. and Sanders, P. (1990), "From Z Specification to Functional Implementations," In *Proceedings of the Z User Workshop*, Oxford, UK, J.E. Nicholls, Ed., Workshops in Computing, Springer-Verlag, Berlin, Germany, 86-112.
- Jia, X. (1994), *ZTC: A type checker for Z: User Guide*, Institute of Software Engineering.

- Kano, N., Seraku, N., Takahashi, F. and Tsuju, S. (1984), "Attractive and Normal Quality," (in Japanese), *Quality*, 14, 2, 39 - 48.
- Kernighan, B.W. and Ritchie, D.M. (1978), *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, USA.
- King, S., Sørensen, I.H. and Woodcock, J. (1988), *Z: Grammar and Concrete and Abstract Syntaxes (Version 2.0)*, Technical Monograph PRG-68, Oxford University Programming Research Group, July 1988, ISBN-0-902928-50-3.
- Knott, R.D. and Krause, P.J. (1988), "An Approach to Animating Z using Prolog," *Report No AI.1*, University of Surrey, Department of Mathematics, Alvey Project SE/065. July 1988.
- Knott, R.D. and Krause, P.J. (1992), "The Implementation of Z specifications using Program Transformation Systems: The SuZan Project," *The Unified Computation Laboratory*, C. Rattray and R.G. Clark, Eds., IMA Conference Series, 35, Oxford University Press, Oxford, UK, 207-220.
- Kowalski, R.A. (1979), "Algorithm = Logic + Control," *Communications of the ACM*, 22, 7, 424 - 435.
- Kowalski, R.A. (1985), "The Relation between logic programming and logic specification," *Mathematical Logic and Programming Languages*, C.A.R Hoare and J.C. Shepherdson, Eds., Prentice-Hall, Englewood Cliffs, New Jersey, USA.
- Kramer, J. and Keng, N. (1988), "Animation of Requirements Specifications," *Software Practice and Experience*, 18, 8, 749 - 774.
- Kuznik, F. (1994), "Blundersat," *Air and Space Smithsonian*, Dec. 1993/ Jan 1994, 41-47.
- Lamport, L. (1985), *L^AT_EX: A Document Preparation System*, Addison-Wesley, Massachusetts, USA.
- Lano, K. (1991), "An Object-oriented Extension to Z," In *Proceedings of the Z User Meeting*, Oxford, UK, J. Nicholls, Ed., *Workshops in Computing*, Springer-Verlag, Berlin, Germany.
- Lano, K. and Haughton, H. (1993), *Object-oriented Specification Case Studies (First Edition)*, Prentice-Hall, Englewood Cliffs, New Jersey, USA.
- Lano, K. (1995), *Formal Object-Oriented Development*, Springer-Verlag, New York, USA.

- Lee, B. (1979), *Introducing Systems Analysis and Design*, Vols 1 and 2, NCC, Manchester.
- Lehman, M. M. (1980), "Programs, Life Cycles and the Laws of Software Evolution," *Proceedings of the IEEE*, 68, 9, 1060-1076.
- Lehmann, T. and Loeckx, J. (1987), "The Specification Language of OBSCURE," In *Proceeding of the 5th Workshop on Specification of Abstract Data Types: Recent Trends in Data Type Specification*, 131-153.
- Leveson, N.G. (1990), "Guest Editor's Introduction: Formal methods in Software Engineering," *IEEE Transactions on Software Engineering*, 16, 9, 929 - 931.
- Logica Inc. (1995), *Formalizer : A Formal Methods support tool for Small Computers*, Logica Cambridge Ltd, UK.
- Lubars, M., Potts, C. and Richter, C. (1993), "A Review of the State of Practise in Requirements Modelling," In *Proceedings of the IEEE International Symposium on Requirements Engineering*, San Diego, California, IEEE Computer Society Press, Los Alamitos, California, USA, 2 – 15.
- Macaulay, L.A. (1996), *Requirements Engineering*, Springer-Verlag, London, UK.
- May, D. (1990), "Use of Formal Methods by a Silicon Manufacturer," *Developments in Concurrency and Communication*, C.A.R.Hoare, Ed., Addison-Wesley, New York, USA, 107-129.
- McCracken, D.D. and Jackson, M.A. (1981), "A Minority Dissenting Position," *Systems Analysis and Design - A foundation for the 80's*, W.W. Cotterman, Ed., 551 – 553.
- McCracken, D.D. and Jackson, M.A. (1982), "Lifecycle Concept Considered Harmful," *ACM Software Engineering Notes*, SE-7, 2, 29 - 32.
- McMenamin, S. and Palmer, J. (1984), *Essential Systems Analysis*, Prentice-Hall, Englewood Cliffs, New Jersey, USA.
- Meira, S.R.L and Cavalcanti, A.L.C. (1991), "Modular Object-oriented Z Specifications," *Z User Meeting 1990*, Workshops in Computing, Springer-Verlag, London, UK, 173-192.
- Microsoft Corp. (1992), *Microsoft Windows Programmer's Reference: Volume 1 Overview*, Microsoft Press, USA.

- Mills, H.D., Dyer, M. and Linger, R.C. (1987), "Cleanroom Software Engineering," *IEEE Software*, 4, 5, 19 - 26.
- Morgan, C. (1987), "Telephone Network," *Specification Case Studies: First Edition*, I. Hayes, Ed., Prentice-Hall, Englewood Cliffs, New Jersey, USA.
- Morgan, C. (1993), "Telephone Network," in *Specification Case Studies: Second Edition*, I. Hayes, Ed., Prentice-Hall, Englewood Cliffs, New Jersey, USA, 31 - 42.
- Morgan C., Robinson K. and Gardinier, P. (1988), "On the Refinement Calculus," *Technical Monograph PRG-70*, Oxford University Computing Laboratory Programming Research Group, Oxford, UK.
- Morrey, I., Siddiqi, J.I.A, and Briggs, J. (1992), "Z Animation in LISP," In *Proceedings of the 5th International Conference on Putting into Practise Methods for Information Systems Design*, Nantes, France.
- Morrey, I., Siddiqi, J. I. A., Buckberry, G., and Hibberd, R. (1994), "Systematic Development of Quality Production Prototypes," In *Proceeding of the IEEE International Conference on Requirements Engineering*, Colorado USA, IEEE Computer Society Press, Los Alamitos, California, USA.
- Morrey, I., Siddiqi, J. I. A., Hibberd, R. and Buckberry, G. (1998), "A Toolset to Support the Construction and Animation of Formal Specifications," *Journal of Systems and Software*, 41 (1998), 147 - 160.
- Moulding, M.R. and Newton, A.R. (1992), "Rapid Prototyping from VDM Specifications using ADA," *IEE Colloquium on Automating Formal Methods for Computer-Assisted Prototyping*, 14th January 1992.
- Mumford, E. and Weir, M. (1979), *Computer Systems in Work Design, The Ethics Method*, Associated Business Press.
- Myopoulos, J., Chung, L., and Nixon, B. (1992), "Representing and Using Nonfunctional Requirements: A Process-Oriented approach," in *IEEE Transactions on Software Engineering*, 18, 6, 483 - 497.
- Naumann, J.D., Davis, G.B. and McKeen, J.D. (1980), "Determining Information Requirements: A Contingency Method for selection of a Requirements Assurance Strategy," *Journal of Systems Software*, 1, 4.

- Naur, P. and Randell, B. (1969), "Software Engineering: Report on a Conference Sponsored by the NATO Science Commission," Garmisch, Germany, 7-11 Oct, 1968. Scientific Affairs Division, NATO, Brussels.
- Nicholls, J.E. (1991), "Domain of Application for Formal Methods", In *Proceedings of the 6th Z User Meeting*, York, December 1991.
- North, N.D. (1990), *An Implementation of Sets and Maps as Miranda Abstract Data Types*, NPL Report DITC 162/90, February 1990.
- O'Neill, G. (1992), "Automatic Translation of VDM Specifications into Standard ML Programs," *The Computer Journal*, 35, 6, 623 - 624.
- Parry, P.W., Ozcan, M.B. and Siddiqi, J. (1995), "The Application of Visualisation to Requirements Engineering," In *Proceedings of the Conference on Software Engineering and its Applications*, France, 699 - 710.
- Place, P.R.H., Wood, W. and Tudball, M. (1990), "Survey of Formal Specification techniques for Reactive Systems," *Software Engineering Institute, CMU/SEI-90-TR-5*. May 1990.
- Pohl, K. (1993), "The three dimensions of Requirements Engineering," In *Proceedings of the 5th International Conference on Advanced Information Systems Engineering (CaiSE '93)*, C. Rolland., F. Bodart. and C. Cauvet, Eds., Springer-Verlag, Paris, 175 - 292.
- Potter, B., Sinclair, J. and Till, D. (1991), *An Introduction to Formal Specification and Z*, Prentice-Hall, Englewood Cliffs, New Jersey, USA.
- Pressman, R.S. (1982), *Software Engineering: A Practitioners Approach*, McGraw Hill Inc, Singapore.
- QSS Inc. (1998), *DOORS Reference*, Quality Systems and Software Inc, 200 Valley Road, Suite 306, Mt. Arlington, NJ 07856, USA.
- Ramesh, B., Stubbs, C., Powers, T., and Edwards, M. (1997), "Requirements Traceability: Theory and Practice," *Annals of Software Engineering*, 3(1997), 397 - 416.
- Rayward-Smith, V.J. (1995), *A First Course in Formal Language Theory 2nd Edition*, McGraw-Hill Book Company, London, UK.

- Reilly, J.P. (1997), "Entity Relationship Approach to Data Modelling," *Software Requirements Engineering 2nd Edition*, R.H. Thayer and M. Dorfman, Eds., IEEE Computer Society Press, Los Alamitos, California, USA, 275 – 285.
- Richardson, D., Aha, S. and O'Malley, T. (1992), "Specification-Based Test Oracles for Reactive Systems," In *Proceedings of the 14th IEEE International Conference on Software Engineering*, Melbourne, Australia, 105-118.
- Ritchie, B. (1993), *Proof with Mural*, Rutherford Appleton Laboratory, Informatics Department, Chilton, Didcot, Oxon, UK.
- Robinson, B. (1994), "Social Context and Conflicting Interests." In *Proceedings of the Second BCS Conference on Information Systems Methodologies*, Edinburgh, UK, British Computer Society, 235 – 249.
- Rolland, C. and Plihon, V. (1996), "Using Generic Method Chunks to Generate Process Model Fragments," In *Proceedings of ICRE Second International Conference on Requirements Engineering*, Colorado, USA, IEEE Computer Society Press, Los Alamitos, California, USA, 173 – 189.
- Ross, D.T. (1977), "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, 3, 1, 16 - 33.
- Royce, W.W. (1970), "Managing the Development of Large Software Systems," In *Proceedings of the IEEE, WESCON*, 1 – 9.
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991), *Object-Oriented Modelling and Design*, Prentice-Hall, EngleWood Cliffs, New Jersey, USA.
- Saiedian, H (1997), "Formal Methods in Information Systems Engineering," *Software Requirements Engineering 2nd Edition*, R.H. Thayer and M. Dorfman, Eds., IEEE Computer Society Press, Los Alamitos, California, USA, 336 – 348.
- Saaltnik, M. (1989), "Z and Eves," In *Proceedings of the Z User Workshop*, Oxford, UK, J. Nicholls, Ed., Workshops in Computing, Springer-Verlag, Berlin, 223 - 242.
- Sampio, A. and Meria, S. (1990), "Modular Extensions to Z," *VDM and Z*, Lecture Notes in Computer Science, 428, Springer-Verlag, Berlin.
- Semmens, L.T., France, R.B. and Docker, T.W.G. (1992), "Integrated Structured Analysis and Formal Specification Techniques," *Computer*, 35, 6.

- Semmens, L.T. and Allen, P. (1991), "Using Yourdon and Z: An Approach to formal Specification," In *Proceedings of the 5th Z user Workshop*, Oxford, UK, J. Nicholls, Ed., Springer-Verlag, Heidelberg, Germany.
- Semmens, L.T. and Allen, P. (1992), "Formalising Yourdon," In *Proceedings of the Methods Integration Workshop*, Leeds, UK, P. Allen, A. Bryant and L. Semmens, Eds., Springer-Verlag, Heidelberg, Germany.
- Sherrell, L.B. and Carver, D.L. (1993), "Z Meets Haskell: A Case Study," In *Proceedings of the 17th Annual International Computer Software & Applications Conference*, 320-326.
- Shlaer, S. and Mellor, S. (1992), *Object LifeCycles : Modelling the world in States*, Yourdon Press. New York, USA.
- Siddiqi, J., Morrey, I., Shaw, S. and Briggs, J. (1991), "Rapid Prototyping of Formal Specifications," In *Proceedings of the 4th International Conference on Software Engineering and its Applications*, Toulouse, France.
- Siddiqi, J. I. A., Morrey, I., Ozcan, M. and Roast, C. (1997), "Towards Quality Requirements via Animated Formal Specifications," *Annals of Software Engineering*, 3(1997), 131 - 155.
- Siddiqi, J. I. A., Morrey, I., Hibberd, R. and Buckberry, G. (1998) "Understanding and Exploring Formal Specifications," *Annals of Software Engineering*, 6(1998), 411 - 432.
- Sommerville, I. (1985), *Software Engineering 2nd Edition*, Addison-Wesley Publishing Company Inc, Wokingham, UK.
- Sommerville, I. and Sawyer, P. (1997), *Requirements Engineering*, John Wiley, Chichester, UK.
- Spanoudakis, G. and Finkelstein, A. (1997), "Reconciling Requirements: A Method for managing interference, inconsistency and conflict," *Annals of Software Engineering*, 3 (1997), 433 - 457.
- Spivey, J.M. (1988), *The FUZZ manual*, Oxford Programming Research Group, Oxford, UK.
- Spivey, J.M. (1989), "An Introduction to Z and Formal Specification," *Software Engineering Journal*, 4,1, 40 - 50.

- Spivey, J.M. (1992), *The Z Notation: A Reference Manual: Second Edition*, Prentice Hall International, Hemel Hempstead, Hertfordshire, UK.
- Tenenbaum, A.M. and. Augenstein, M.J. (1986), *Data structures using Pascal, 2nd Edition*, Prentice-Hall, Englewood Cliffs, New Jersey, USA.
- Thimbleby, H. (1990), *User Interface Design*, ACM Press, New York, USA.
- Turner, D.A. (1985), "Functional Programs as Executable Specifications," *Mathematical Logic and Programming Languages*, C.A.R. Hoare and J.C. Shepherdson, Eds., Prentice-Hall, Englewood Cliffs, New Jersey, USA.
- Vadera, S. and Meziane, F. (1997), "Tools for Producing Formal Specification: A view of current architectures and future directions," *Annals of Software Engineering*, 3(1997), 273 - 290.
- Valentine, S.H. (1995), "The Programming Language Z—," *Information and Software Technology*, 37, 5, 293 - 301.
- Valusek, J.R. and Fryback, D.G. (1985), "Information Requirements Determination: Obstacles within, among and between participants," In *Proceedings of the End-User Computing Conference*, ACM. Press, New York, USA.
- Vienneau, R. (1991), *An Overview of Object-Oriented Design*, Data & Analysis Centre for Software, Apr 30, 1991.
- Vienneau, R. (1997), "A Review of Formal Methods," *Software Requirements Engineering 2nd Edition*, R.H. Thayer and M. Dorfman, Eds., IEEE Computer Society Press, Los Alamitos, California, USA, 324 – 335.
- Verheijen, G.M. and Van Bekkum, J. (1982), "NIAM: An information analysis method," *Information Systems Design Methodologies: A comparative view*, T.W. Olle, Ed., North Holland.
- Ward, P. and Mellor, S. (1985), *Structured Development for Real-Time Systems*, Prentice-Hall, Englewood Cliffs, New Jersey, USA.
- Wassermann, A.I., Freeman, P. and Porcella, M. (1983), "Characteristics of Software Design methodologies," *Information System Design methodologies: A Feature Analysis*, T.W. Olle, Ed., North Holland.
- Wassermann, A.I. and Shewmake, D.T. (1985), "The role of prototypes in the User Software Engineering (USE) Methodology," *Advances in Human-Computer Interaction, 1*, H.R. Hartson, Ed., Norwood, New Jersey, USA.

- Weinburg, V. (1978), *Structured Analysis*, Prentice-Hall, New York, USA.
- West, M. and Eaglestone, B. (1992), "Software Development: Two approaches to Animation of Z specifications using Prolog," *IEE Software Engineering Journal*, 7, 4, 264-276.
- Wilensky, R. (1986), *Common LISPcraft*, W.W. Norton and Company, New York, USA.
- Wing, J.M. (1990), "A specifier's introduction to Formal Methods," *Computer*, 23, 9, 8 - 23.
- Wirth, N. (1971), "Program Development by Stepwise Refinement," *Communications of the ACM*, 14, 4, 221-227.
- Wordsworth, J.B. (1987), *A Z Development Method*, Technical Report, IBM UK Laboratories Ltd, Hursley Park, UK.
- Wordsworth, J.B. (1989), "A Z Development Method," In *Proceedings of the Workshop on Refinement*, Milton Keynes, UK, The Open University.
- Wulf, W.A., Shaw, W., Hilfinger, P.N. and Flon, L. (1981), *Fundamental Structures of Computer Science*, Addison-Wesley Publishing Company, Massachusetts, USA.
- York Software Engineering. (1991), *CADIZ - The CADiZ Tutorial*, Version 1.0, York Software Engineering Ltd, York, UK.
- Yourdon, E. and Constantine, L. (1979), *Structured Design*, Yourdon Press, New York, USA.
- Yourdon, E. (1989), *Modern Structured Analysis*, Prentice-Hall, Englewood Cliffs, New Jersey, USA.
- Zin, A.M. (1993), *ZFDSS: A Formal Development Support System based on the Liberal Approach*, Ph.D Thesis, University of Nottingham, UK.
- Zin, A.M. and Foxley, E. (1991) "Automatic Program Quality Assessment System," In *Proceedings of the IFIP Conference on Software Quality*, S.P. University, Vidyanagar, India.
- Zultner, R.E. (1991), "Quality Function Deployment (QFD) for Software: Structured Requirements Exploration". *Total Quality management for Software*, 297 – 319.
- Zultner, R.E. (1993), "TQM for technical Teams". *Communications of the ACM*, 36, 10, 79 – 91.

Appendix I: LL(*k*) Grammar for the Z Notation

This appendix contains the re-written LL(*k*) grammar for the Z notation used in the implementation of the TranZit Analyser Subsystem (TAS). It is based on the grammar originally devised by Spivey (1992). The re-written LL(*k*) grammar removes direct-left recursion from Spivey's original grammar and ensures that the set of first symbols in each rule is pairwise disjoint. The grammar contains *meta-symbols* as defined in Table 7.

Grammar Meta-Symbols	Meaning
' '	'or'
[<i>X</i>]	Symbol <i>X</i> is optional
(<i>X</i>)	factoring of Symbol <i>X</i>
{ <i>X</i> }	zero or more occurrences of symbol <i>X</i> (<i>iteration</i>)
<i>X</i> , ... , <i>X</i>	One or more instances of Symbol <i>X</i> , separated by commas (<i>iteration</i>)
<i>X</i> ... <i>X</i>	One or more instances of Symbol <i>X</i> , with no separators (<i>iteration</i>)

Table 7: Table of Meta-Symbols for the LL(*k*) Grammar

Newline symbols (NL) are not treated as white-space characters in the Z grammar. Newline symbols may only occur where specified in the grammar, or surrounding the symbols defined in Table 8:

$ \begin{array}{c} ; : , \bullet = = \hat{=} :: = \in \wedge \vee \Rightarrow \Leftrightarrow \times \setminus \uparrow \\ \end{array} $
--

Table 8: Symbols Which Can Be Surrounded by NL Characters

Spaces are not considered significant, except where they serve to separate one symbol from another. Non-terminal symbols are identified as beginning with an upper-case character in *italics*; all terminal symbols are in bold lower case.

Terminal Symbols are defined in Table 9.

Specification ::= Paragraph NL ... NL Paragraph

Paragraph ::= [Ident , ... , Ident]
| *Axiomatic-Box*
| *Schema-Box*
| *Generic-Box*
| *Schema-Name [Gen-Formals] ≅ Schema-Exp*
| *Ident (::= Ident | ... | Ident*
| | *In-Gen Ident == Expression*
| | *[Gen-Formals] == Expression*
| |)
| *(Op-Name) [Gen-Formals] == Expression*
| *Pre-Gen Ident == Expression*
| *Predicate*

Axiomatic-Box ::= [*Decl-Part* [
| *Axiom-Part*]

Schema-Box ::= *Schema-Name [Gen-Formals]* _____
| *Decl-Part* [
| *Axiom-Part*] _____

Generic-Box ::= *Schema-Name [Gen-Formals]* _____
| *Decl-Part* [
| *Axiom-Part*] _____

Decl-Part ::= *Basic-Decl Sep ... Sep Basic-Decl*

Axiom-Part ::= *Predicate Sep ... Sep Predicate*

Sep ::= ; | NL

Schema-Exp ::= \forall *Schema-Text* • *Schema-Exp*
| \exists *Schema-Text* • *Schema-Exp*
| *Schema-Exp-1*

Schema-Exp-1 ::=
(
 [*Schema-Text*]
 | *Schema-Ref*
 | \neg *Schema-Exp-1*
 | **pre** *Schema-Exp-1*
 | (*Schema-Exp*)
)
[(
 (\wedge | \vee | \Rightarrow | \Leftrightarrow | \uparrow | \S) *Schema-Exp-1*
 | \backslash (*Decl-Name*, ... , *Decl-Name*)
)]

Schema-text ::= *Declaration* [\backslash *Predicate*]

Schema-Ref ::= *Schema-Name Decoration* [*Gen-Actuals*]
[*Renaming*]

Renaming ::= [*Decl-Name* / *Decl-Name* , ...,
Decl-Name / *Decl-Name*]

Declaration ::= *Basic-Decl ; ... ; Basic-Decl*

Basic-Decl ::= *Ident , ... , Ident : Expression*
| *Schema-Ref*
| *Op-Name , ... , Op-Name : Expression*

Predicate ::= \forall *Schema-Text* • *Predicate*
| \exists *Schema-Text* • *Predicate*
| **let** *Let-Def ; ... ; Let-Def* • *Predicate*
| *Predicate-1*

Predicate-1 ::=
(
| *Pre-Rel Expression*
| **pre** *Schema-Ref*
| **true**
| **false**
| \neg *Predicate-1*
| (*Expression Rel Expression Rel ...*
| *Rel Expression*)
| (*Predicate*)
| *Expression Rel Expression Rel ...*
| *Rel Expression*
)
[(
| (\wedge | \vee | \Rightarrow | \Leftrightarrow) *Predicate-1*
)]

Rel ::= = | \in | *In-Rel Decoration*

Let-Def ::= *Var-Name == Expression*

Expression-0 ::=

- | λ *Schema-Text* • *Expression*
- | μ *Schema-Text* [• *Expression*]
- | **let** *Let-Def*; ... ; *Let-Def* • *Expression*
- | *Expression*

Expression ::=

- | *Expression-1* { \times *Expression-1* }
- | [*In-Gen Decoration* *Expression*]
- | **if** *Predicate* **then** *Expression* **else** *Expression*

Expression-1 ::= (

- | *Pre-Gen Decoration* *Expression-3*
- | (**P** | **U** | **N**) *Expression-3*
- | - *Decoration* *Expression-3*
- | *Expression-3* ({ *Expression-3* }
- | *Post-Fun Decoration*
- | (*Expression-0*)
-)

) [*In-Fun Decoration* *Expression-1*]

Expression-3 ::= (

- | *Word Decoration* [*Gen-Actuals*]
- | *Schema-Ref* [([*Ident* / *Ident*]
- | [*Gen-Actuals*]
-)]
- | *String*
- | ((*Op-Name*
- | λ *Schema-Text* • *Expression*
- | μ *Schema-Text* [• *Expression*]
- | *Expression*
-))
- | *Number*
- | { [(
- Schema-Text* [• *Expression*]
- | *Expression* { , *Expression* }
-)] }
- | \langle [*Expression* { , *Expression* }] \rangle
- | Θ *Schema-Name Decoration* [*Renaming*]
- | \llbracket [*Expression* { , *Expression* }] \rrbracket

) [• *Var-Name*]

Ident ::= *Word Decoration*

Decl-Name ::= *Op-Name* | *Ident*

Var-Name ::= *Ident* | (*Op-Name*)

Op-Name ::= *_ In-Sym Decoration _* | *Pre-Sym Decoration _*
| *_ Post-Sym Decoration* | *_ (_) Decoration*
| *- Decoration*

In-Sym ::= *In-Gen* | *In-Fun* | *In-Rel*

Rel ::= = | \in | *In-Rel*

Pre-Sym ::= *Pre-Gen* | *Pre-Rel*

Post-Sym ::= *Post-Fun*

Decoration ::= [*Stroke* , ... , *Stroke*]

Gen-Formals ::= [*Ident* , ... , *Ident*]

Gen-Actuals ::= [*Expression* , ... , *Expression*]

String ::= “ [*char* ... *char*] “

Terminal Symbol Sets	Symbols
<i>Word</i>	Undecorated name or special symbol
<i>Char</i>	Any character
<i>Stroke</i>	Single decoration ‘ , ? , !
<i>Schema-Name</i>	Word used as a schema name
<i>In-Fun</i>	Priority 1: \mapsto Priority 2: .. Priority 3: + - \cup \setminus \cap \oplus Priority 4: * div mod \cap \uparrow \circ Priority 5: \oplus # Priority 6: \triangleright \triangleright \triangleleft \triangleleft
<i>Post-Fun</i>	* - + - - -
<i>In-Rel</i>	\notin \neq \subset \subseteq \langle \rangle prefix suffix in partition \leq \geq
<i>Pre-Rel</i>	disjoint
<i>In-Gen</i>	\mapsto \rightarrow \rightsquigarrow \rightsquigarrow \rightarrow \rightarrow \rightsquigarrow \rightsquigarrow \leftrightarrow
<i>Pre-Gen</i>	Id F seq iseq bag
<i>Number</i>	{0 .. 9}

Table 9: Table of Terminal Symbols for LL(k) Grammar

Appendix II: Context-Free Grammar of the ZAL Language

This appendix contains the CFG of the ZAL Language, as derived for the purposes of transforming a Z notation specification to the ZAL language. The grammar is described using *meta-symbols* as defined in Table 10.

Grammar Meta-Symbols	Meaning
' '	'or'
[X]	Symbol X is optional
(X)	factoring of Symbol X
{X}	zero or more occurrences of symbol X (<i>iteration</i>)
<i>X</i> , ... , <i>X</i>	One or more instances of Symbol X, separated by commas (<i>iteration</i>)
<i>X</i> ... <i>X</i>	One or more instances of Symbol X, separated by spaces (<i>iteration</i>)

Table 10: Table of Meta-Symbols for the ZAL Grammar

Spaces are not considered significant, except where they serve to separate one symbol from another. Non-terminal symbols are identified as beginning with an upper-case character in *Italics*. All terminal symbols are in bold lower case.

In common with Spivey's (1992) nomenclature, each production for which a *binding power* is relevant is marked with either an upper case 'L' or 'R' indicating association to the left or right respectively. Unary symbols are marked with a 'U'. To make explicit the mapping between ZAL functions and Spivey's original organisation of the mathematical toolkit, symbolic group names have been retained from Spivey's original grammar.

The set of terminal symbols for this grammar is shown in Table 11.

Generator ::= *Ident Expression Generator*

Binary-Relation ::= (*Rel Expression Expression*)
| (*Rel Expression Binary-Relation*)

Predicate-1 ::= (*disjoint Expression*)
| (*disjoint-dis Expression*)
| (*not Predicate-1*) U
| *Binary-Relation*
| (*and Predicate-1 Predicate-1*) L
| (*or Predicate-1 Predicate-1*) L
| (*imply Predicate-1 Predicate-1*) R

Rel ::= *eqz* | *mem* | *In-Rel*

Expression ::= *Expression-1*
| (*if Predicate Expression Expression*)

Expression-1 ::= (*In-Gen Expression-1 Expression-1*) R
| (*card Expression-4*)
| (*Pre-Gen Expression-4*)
| *Expression-3*
| (*rel-image Expression-4 Expression*)
| (*In-Fun Expression-1 Expression-1*) L
| # (*Expression-1 Expression 1*)

Expression-3 ::= *Expression-3 Expression 4*
| *Expression-4*
| (*applyz Func-Name Var-name*)

Renaming ::= (*schema-rename Schema-Name Renames*)

Renames ::= *Renames* | *ident ident*

Expression-4 ::= *Var-Name*
| *'Literal*
| *Number*
| *Set-Exp*
| *(execute Schema-Name [renaming])*
| *(inverse Expression-4)*
| *< [Expression ... Expression] >*

Set-Exp ::= { *Expression ... Expression* }
| *(mksi GenVars Generator 'Expression)*

Decl-Name ::= *Ident*

Ident ::= *Word*

Pre-Sym ::= *Pre-Gen | Pre-Rel*

In-Sym ::= *In-Gen | In-Fun | In-Rel*

Var-Name ::= *Ident*

GenVars ::= *'Ident GenVars*

Terminal Symbol Sets	Symbols
<i>Word</i>	Undecorated identifier name
<i>Char</i>	Any character
<i>Func-Name</i>	Word used as a function name
<i>Schema-Name</i>	Word used as a schema name
<i>Literal</i>	Word used as a literal name introduced by Free Type definition
<i>In-Fun</i>	Priority 1: mks Priority 2: + - unionz setsub appendz Priority 3: * floor mod inter rel-compose Priority 4: override Priority 5: domsub domres ransub ranres
<i>In-Rel</i>	Subset psubset not-mem neqz \< \> \<= \>=
<i>Pre-Rel</i>	disjoint
<i>In-Gen</i>	makemap
<i>Pre-Gen</i>	Powerset intersect-dis union-dis
<i>Number</i>	{0 .. 9}

Table 11: Table of Terminal Symbols for the ZAL Grammar

Appendix III : A Review of Current Requirements Engineering Toolsets

As part of this project, it is instructive to review other well-documented toolsets currently available, which address similar problems in requirements engineering to those addressed by the REALiZE toolset. The current state-of-the-art is summarised in Table 12 below.

This table gives general details of other tools available that have been documented, and identifies their associated capabilities. In addition, Table 13 defines the corresponding capabilities of the REALiZe toolset, for the purposes of comparison with the tools described in Table 12.

Tool Name	Application	Language Basis	Language support <i>S = Syntax Check</i> <i>G = Graphical Language Support</i> <i>P = Proof Support</i> <i>R = Refinement Support</i> <i>T = Type Check</i> <i>C = Test Case Generation</i>	Interface type/Host Platform	Animation Support/ Code Generation	Contact
Atelier B	The B method is used to develop critical software components as well as to validate critical system specifications	B method/ Abstract Machine Notation	S/G/P/R/T	Sun Sparc station running SunOS 4.1.x or Solaris 2.x HP 9000 station running HP-UX 9.x PC running Linux 2.x	Yes	atelierb.aix@steria.fr I
B- Toolkit	These integrated tools support software development ranging from specification, through design and coding to maintenance	Abstract Machine Notation (AMN)	S/G/P/T/R	IBM/RS6000 running AIX Sun Sparc running SunOS 4.1.x. Solaris 5.x. Dec Alpha running OSF1 Silicon Graphics running IRIX PC running Linux	Yes	info@b-core.com
CADIZ	Analysis of and investigation of properties of Z specifications.	Z	S/G/T/P	UNIX with X (Sun Solaris-2.5; SGI IRIX-5; 486 Linux).	Some	ian@cs.york.ac.uk

CADP toolbox	The CADP toolbox is dedicated to the efficient compilation, simulation, formal verification, and testing of descriptions written in the ISO language LOTOS [ISO standard 8807].	LOTOS	S/G/T/P/R	Sun 3, Sun4 under SunOS 4.1.*, Sun4 under Solaris 2.* The X-windows system (at least X11R5 or Openwin) The "Ghostview" software	Yes	Hubert.Garav el@inria.fr
Centaur-VDM environment	Graphical tool for manipulating specifications written in BSI VDM-SL	VDM-SL	S/G/R/C	Sun SPARC, X11 version R4.	No	facon@cnam.fr
DisCo	Specification of distributed reactive systems.	DisCo Language	G/P	Sun SPARC, SunOS 4.1 with Open Windows	Yes	pk@cs.tut.fi
DST-fuzz/DST-Z Toolbox	A set of tools to supply syntax checking and type checking. LaTeX based pretty printing for Z specifications.	Z	S/G/T/C	HP-UX 9.0, SunOS 4.1.x, Solaris 2.2. SoftBench required	Yes	hoercher@vst.vossloh.de
Formaliser	Formaliser is a syntax-directed editor and type checker for Z	Z ZEST	S/G/T	Windows 3.1, Windows '95	No	stepneys@iogica.com
Formooz	Formooz supports the development of formal specifications written in MooZ, an OO extension of Z	MooZ	S/G/T	Sparcstation under SunOS with OpenWindows and LaTeX	No	formooz@di.ufpe.br
ICOS	Specification, simulation, verification and synthesis of reactive systems.	PTL, Symbolic Timing Diagrams VHDL Code generation	S/G/P/T	UNIX and Tcl/Tk	Yes	Karsten.Lue th@Informatik.Uni-Oldenburg.DE

IFAD VDM-SL Toolbox	The Toolbox includes a syntax checker, static semantic checker, pretty printer generating LaTeX output. In addition to this it contains a debugger and an interpreter which can execute all executable constructs of VDM-SL	ISO VDM-SL	S/T	Sun Sparc running SunOS 4.1.x or SunOS 5.3 (Solaris 2.3). HP 9000/700 architecture running HP-UX 9.0x Silicon Graphics running IRIX 5.3/6.2 PC/386, 486 or compatible running Linux	Yes	toolbox@i.fad.dk
LOTOS	The LOTOS toolbox contains a number of tools supporting the specification and implementation of LOTOS specifications	LOTOS	S/G	Sun 3, Sun 4, SunOS 16 Mb memory, 35 Mb disk; Hp, HP Unix, 16 Mb memory, 35 Mb disk	Code generation in C	vdvloedt@ita.nl
Mathias	Helps with drafting of a formal specification (e.g. in Z) as the specification itself can be animated and test cases tried out.	Z/ PROLOG	S/G	Any PROLOG system	Yes	R.Knott@su.rrey.ac.uk
Pet Dingo	Tool provides a mechanism to generate a formal system description in ESTELLE to a distributed implementation framework in C++.	Estelle	S	Sun SPARC, 4Mb memory, 25Mb disk Sun OS 4.0x, X11, GNU C++ 1.xx	Yes	National Institute of Standards and Technology

PiZA	PiZA allows Z to be used for animation and rapid prototyping. It also acts as a front end to the LaTeX typesetting language and other tools such as CADiZ, fuzz and ZTC.	Z	S/G	Quintus Prolog 3.2 on UNIX is required to compile the tool	Yes	M.A.Hewitt@pascal.dra.hmg.gb
Proof Power	ProofPower supports document preparation, syntax checking, type checking and formal proof development using Higher Order Logic and/or the Z language	Z, HOL	S/T/P	Sun SPARC, 50Mb disk	No	rda@win.icl.co.uk
SCR Toolset	The toolset supports the creation of SCR requirements specs which specify the required, externally-visible behaviour of software and systems	SCR/NRL/A7 function tables	S/G/T	Sun's SunOS and Solaris, HP's UNIX, DEC's Alpha UNIX	Yes	labaw@itd.nrl.navy.mil
SpecBox	Development of formal software specifications and designs	VDM-SL	S	PC 386 or later running MS DOS 3.2 or later; or Sparcstations running SunOs 4.1 or later	No	pkdf@adelard.co.uk
VDM Through Pictures	be able to work in both graphical and textual form in capturing a formal specification	VDM-SL	S/T/G	SunOS with Software thru Pictures	No	jdick@comlab.ox.ac.uk
Visualizer	Tool for creating Z specifications using icons.	Z	S/G	NeXTSTEP 3.3	No	c.yap@dcs.shef.ac.uk
Venus	Venus provides an	OMT Class	S/T/G	Sun-4 and HP-	Yes	toolbox@ifa

	environment for graphical (in terms of the diagrams used with OMT) and formal (in terms of VDM++, an object-oriented extension of VDM-SL) specification of object-oriented, concurrent systems	diagrams and VDM++		9000/700		<u>d.dk</u>
ZOLA	Zola is an integrated support tool for the Z specification language, providing automated assistance for all stages of the specification construction, proving and maintenance process	Z	S/G/T/P	Zola requires a Sun workstation	No	<u>fms@ist.c</u> <u>o.uk</u>
Z Browser	Besides displaying Z paragraphs as they appear in printed form, Z Browser has rich help for the entire Z Notation which covers both the ZRM and ZRM 2nd Edition	Z	S/G	Windows 3.1x, 95, NT	No	<u>lmikusia</u> <u>@ingr.co</u> <u>m</u>

Table 12: Review of Requirements Engineering Toolsets

In order to make a comparison with the tools listed above, it is necessary to express the capabilities of the REALiZE toolset in similar terms, as shown in Table 13:

Tool Name	Application	Language Basis	Language support S = Syntax Check G = Graphical Language Support P = Proof Support R = Refinement Support T = Type Check C = Test Case Generation	Interface type/Host Platform	Animation Support/ Code Generation	Contact
REALIZE	Integrated toolset comprising TranZit Z editor, Checker and Transformation Engine, ZAL Animation Environment and ViZ Object-oriented visualisation engine	Z	S/G/T	Windows 3.1/95/98 and NT. Allegro Common LISP environment required by ZAL.	Yes	i.c.morrey@shu.ac.uk

Table 13: REALiZE Toolset Capabilities

Appendix IV: TranZit User Questionnaire Results

This appendix contains details of the TranZit User questionnaire and associated results. The questionnaire is divided into four sections dealing with population statistics, requirements engineering, the Z notation and TranZit respectively. The questions are colour coded to the right of each table, with the associated results shown in the bar graph. The height of the bar graph indicates the number of respondents as labelled on the y-axis.

Appendix IV-1: Population Statistics

This section presents two tables of data associated with the types of people who answered the questionnaire.

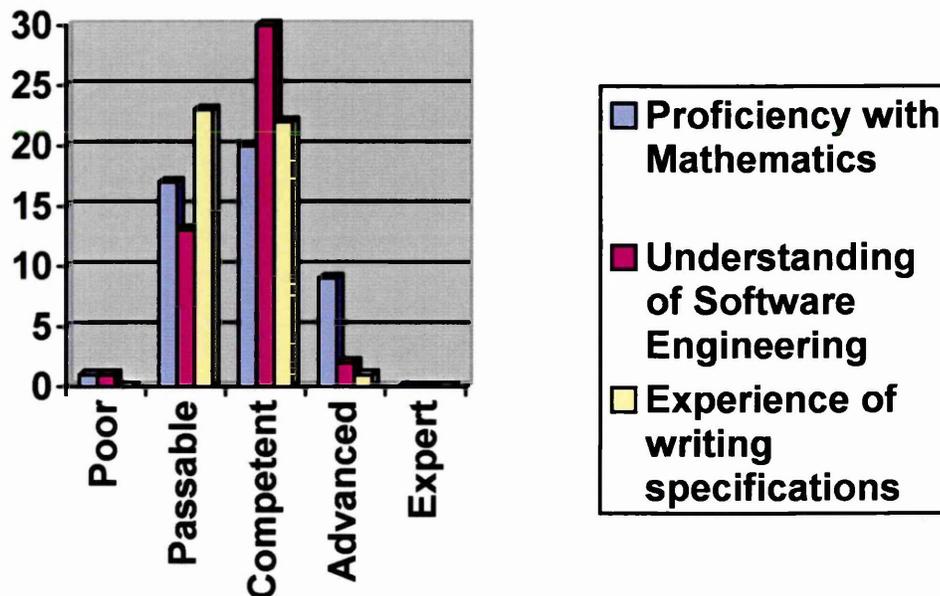


Table 14: Questionnaire Results: User Experience I

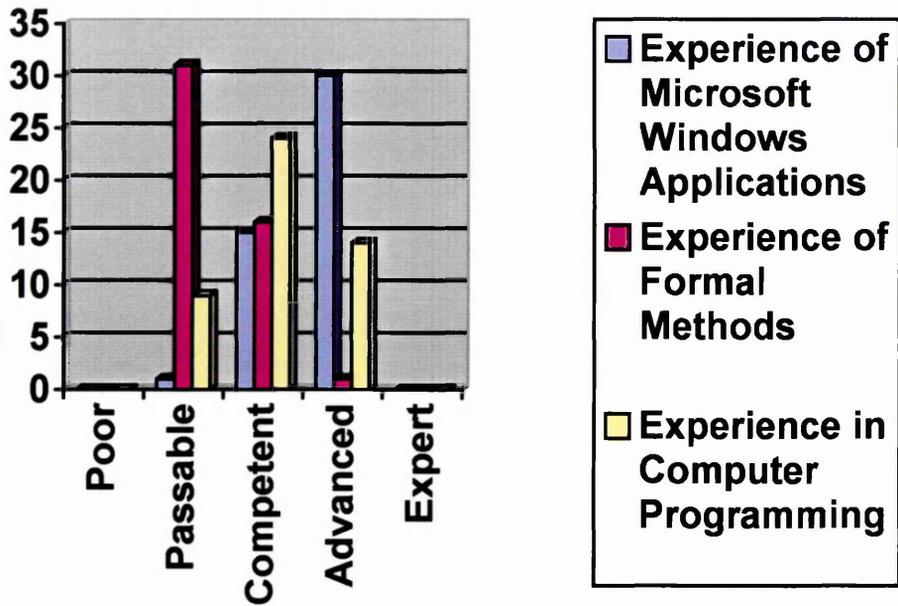


Table 15: Questionnaire Results: User Experience II

Appendix IV-II: About Requirements Engineering

This section presents three tables of data associated with opinions on requirements engineering.

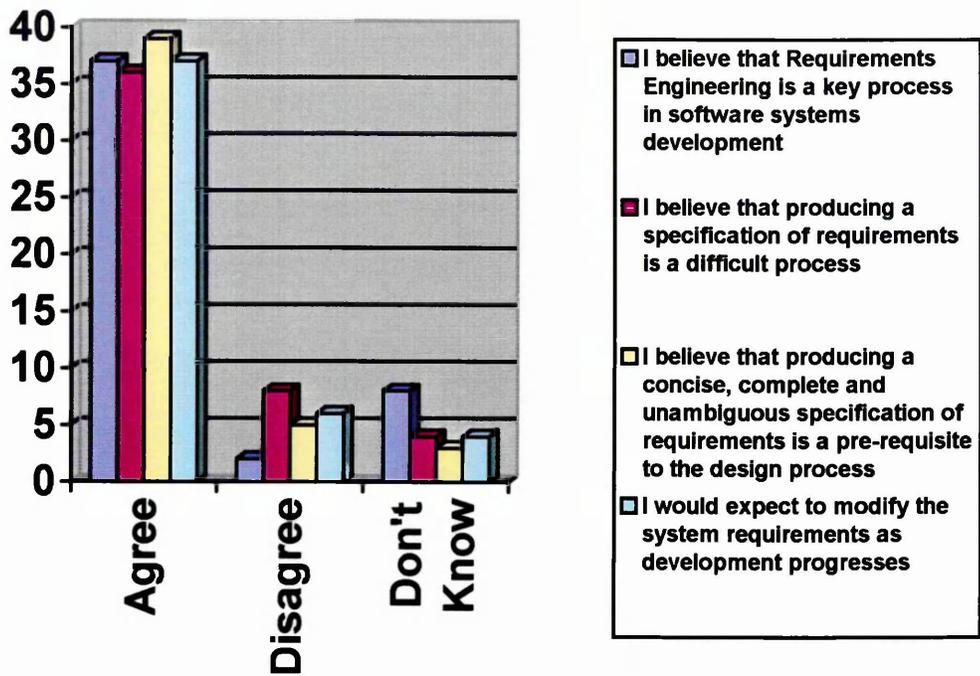


Table 16: Questionnaire Results: Opinions on Requirements Engineering I

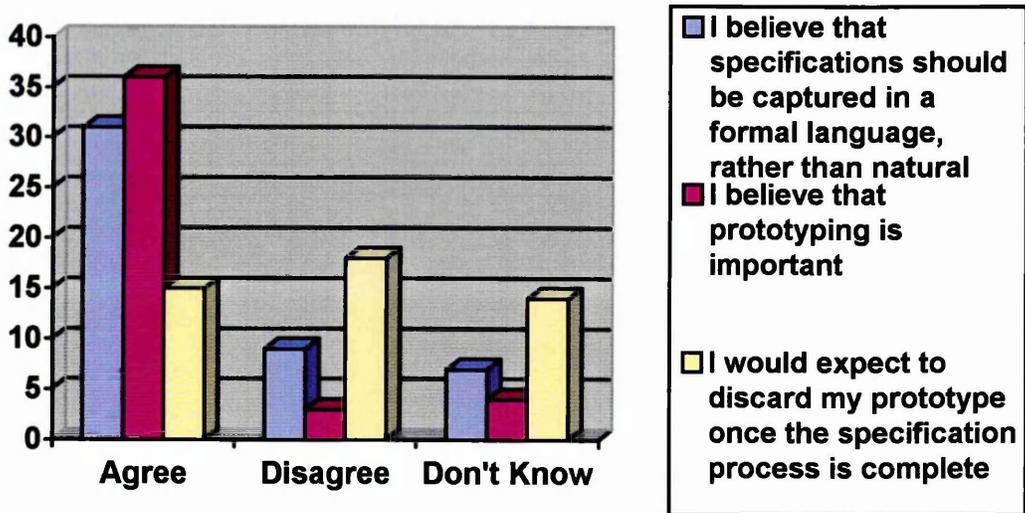


Table 17: Questionnaire Results: Opinions on Requirements Engineering II

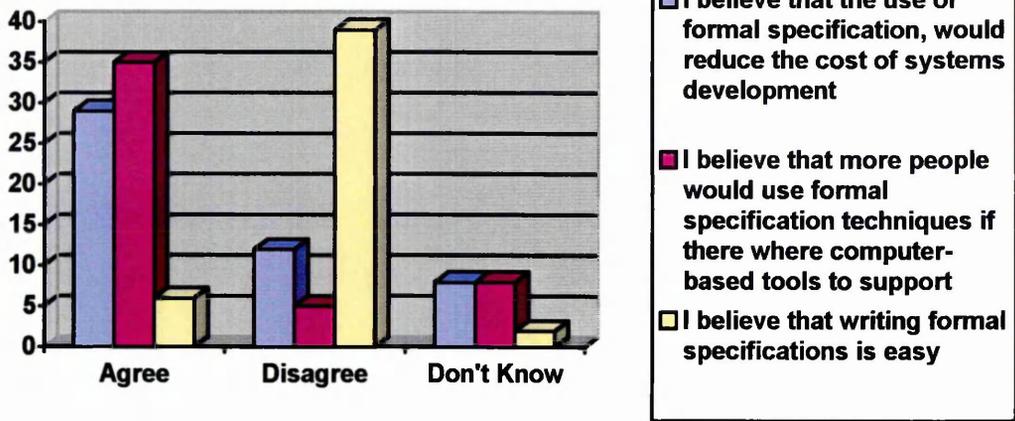


Table 18: Questionnaire Results: Opinions on Requirements Engineering III

Appendix IV-III: About the Z Notation

This section presents three tables of data associated with opinions on the Z notation.

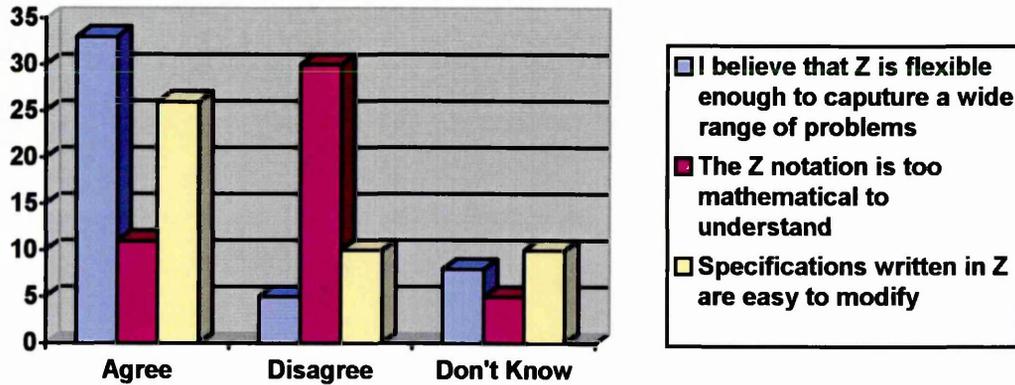


Table 19: Questionnaire Results: Opinions on the Z Notation I

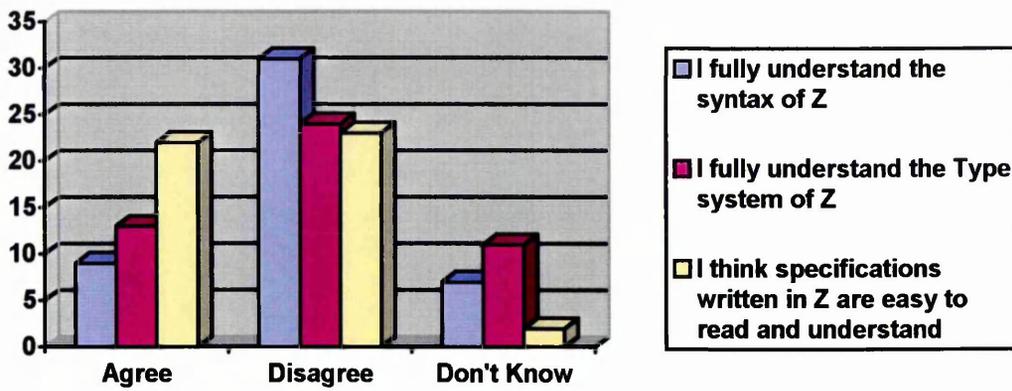


Table 20: Questionnaire Results: Opinions on the Z Notation II

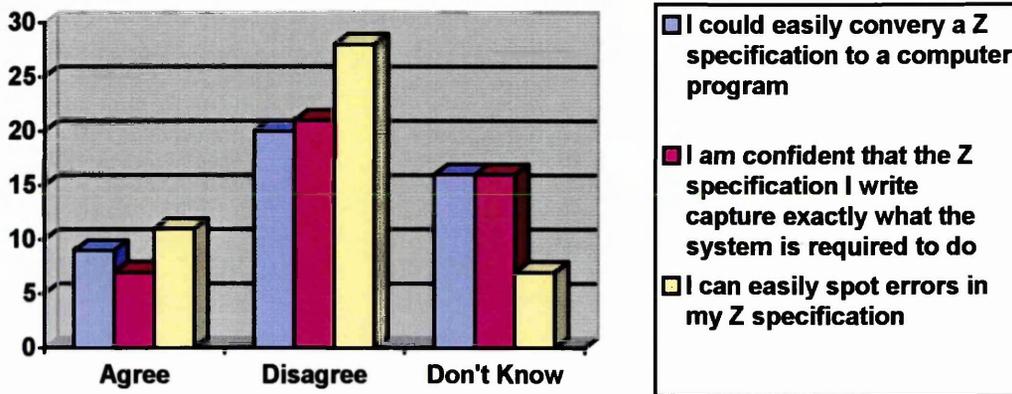


Table 21: Questionnaire Results: Opinions on the Z Notation III

Appendix IV-IV: About TranZit

This section presents four tables of data associated with opinions on the TranZit Tool.

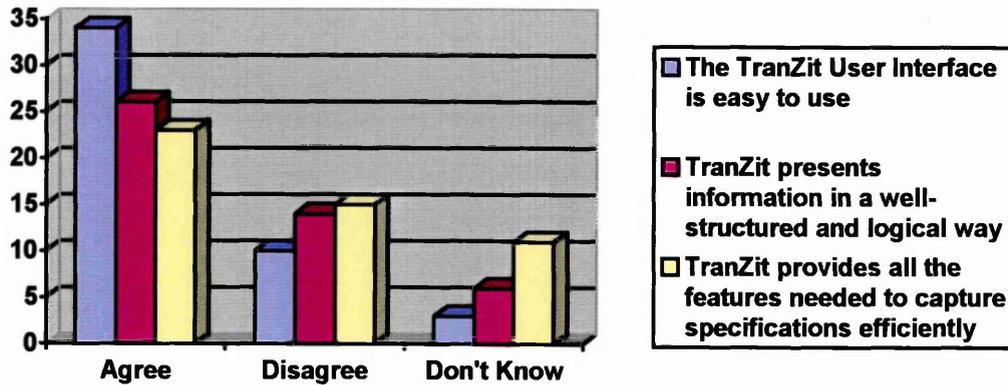


Table 22: Questionnaire Results: Opinions on the TranZit Tool I

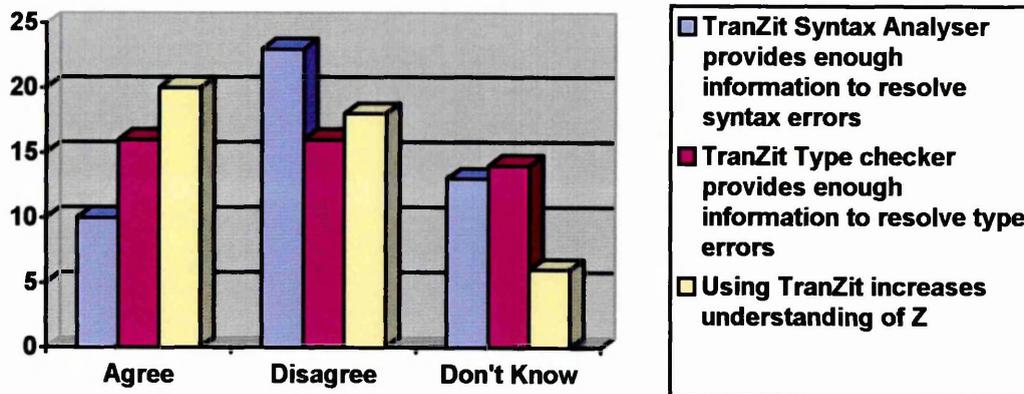


Table 23: Questionnaire Results: Opinions on the TranZit Tool II

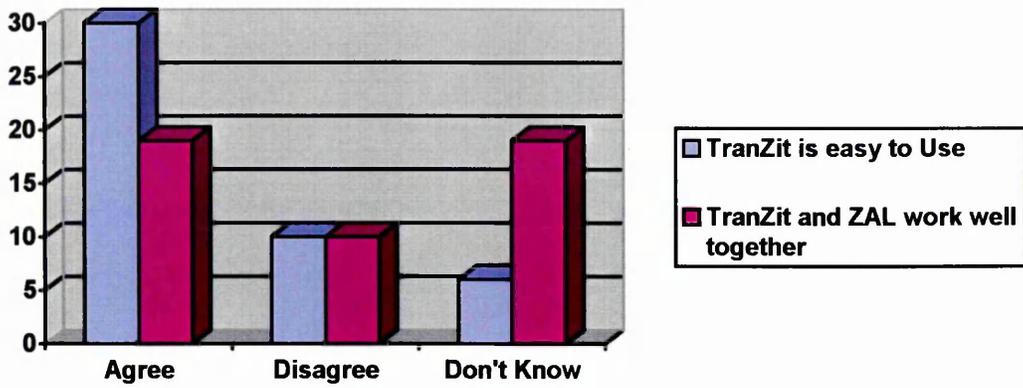


Table 24: Questionnaire Results: Opinions on the TranZit Tool III

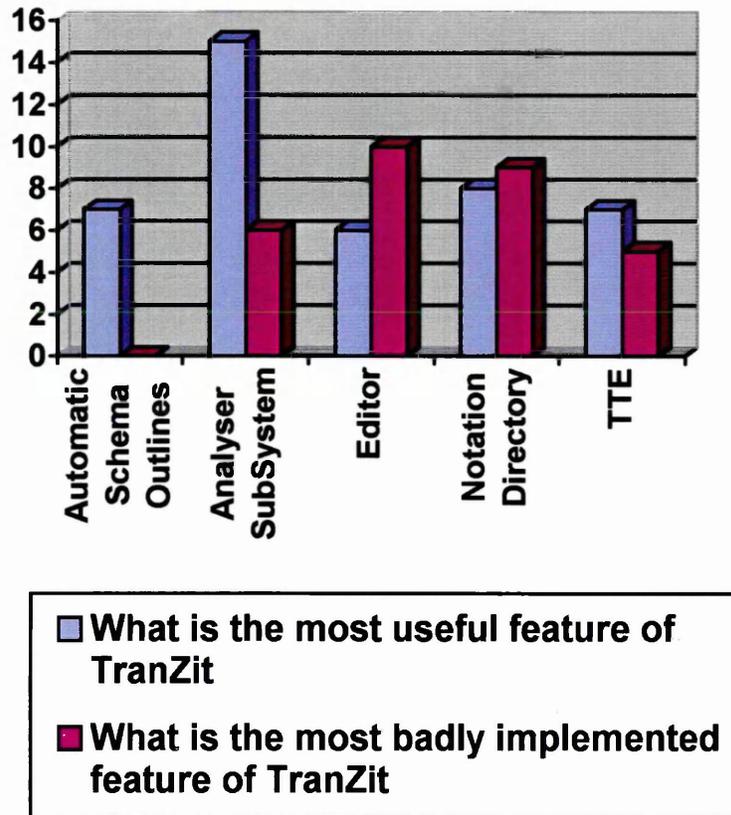


Table 25: Questionnaire Results: Opinions on the TranZit Tool IV

Appendix V: Publication History

Table 26 contains details of the history of publications associated with this project.

Reference	Publication
Morrey [93a]	Morrey, I., Siddiqi, J. I. A., Hibberd, R. and Buckberry, G. "Use of Formal Specification Tools to teach Formal Methods," <i>Fifteenth International Conference "Information Technology Interfaces", ITI' 93</i> , Pula, Croatia, June 1993.
Morrey [93b]	Morrey, I., Siddiqi, J. I. A., Hibberd, R. and Buckberry, G. "Use of a specification construction and animation tools to teach formal methods," <i>IEEE Compsac 93, The Seventeenth Annual International Computer Software and Applications Conference</i> , Phoenix, Arizona, USA, November 1993.
Siddiqi [93]	Siddiqi, J. I. A., Morrey, I., Buckberry, G. and Hibberd, R. "Towards Case Tools for Proto-typing Z specifications," <i>IEEE. Case 93, Sixth International Workshop on CASE</i> , National University of Singapore, Singapore, July 1993.
Morrey [94]	Morrey, I., Siddiqi, J. I. A., Buckberry, G. and Hibberd, R. "Systematic Development of Quality Production Prototypes," <i>IEEE International Conference on Requirements Engineering</i> , Colorado, USA, April 1994.
Siddiqi [95]	Siddiqi, J. I. A. and Morrey, I. "A Toolset to support a software engineering strategy for AI development," <i>IEEE International conference on Tools with AI</i> , Washington, USA, November 1995.
Siddiqi [96]	Siddiqi, J. I. A., Morrey, I. and Hibberd, R. "A Toolset to Support the Formal Specification of AI Systems," <i>1996 Florida AI Research Symposium</i> , Florida, USA. May 1996.
Siddiqi [97]	Siddiqi, J. I. A., Morrey, I., Ozcan, M. and Roast, C. "Towards Quality Requirements via Animated Formal Specifications," <i>Annals of Software Engineering</i> , 3(1997), 131 - 155.
Ozcan [98]	Ozcan, M., Parry, P., Morrey, I. and Siddiqi, J. I. A. "Requirements Validation Based on the Visualisation of Executable Formal Specifications," <i>Proc. 22nd Annual International Computer Software and Application Conference (COMPSAC 98)</i> , Vienna, Austria, 1998, 381-386.
Morrey [98]	Morrey, I., Siddiqi, J. I. A., Hibberd, R. and Buckberry, G. "A Toolset to Support the Construction and Animation of Formal Specifications," <i>Journal of Systems and Software</i> , 41 (1998), 147 - 160.
Siddiqi [98]	Siddiqi, J. I. A., Morrey, I., Hibberd, R. and Buckberry, G. "Understanding and Exploring Formal Specifications," <i>Annals of Software Engineering</i> 6 (1998), 411 - 432

Table 26: Table of Publications Associated with this Project

For further details, visit the project web page at <http://kingfisher.cms.shu.ac.uk/~req-eng/>.

Appendix VI: Glossary of Abbreviations

4GL	Fourth Generation Language
ADT	Abstract Data Type
ATM	Automated Teller Machine
BNF	Backus-Normal Form
CASE	Computer-Aided Software Engineering
CFG	Context-Free Grammar
CFL	Context-Free Language
CICS	Customer Information and Control System
CSP	Communicating Sequential Processes
DDE	Dynamic Data Exchange
ETB	Expression Type Builder
FSA	Finite State Automata
GUI	Graphical User Interface
HOL	Higher Order Logic
HOQ	House of Quality
IEEE	Institute of Electrical and Electronic Engineers
IT	Information Technology
JSD	Jackson Structured Design
MDI	Multiple Document Interface
MS	Microsoft Corporation
NASA	North American Space Agency
NFR	Non-Functional Requirement
NFSA	Non-deterministic Finite State Automata
NPDA	Non-deterministic Pushdown Automata
OMT	Object Modelling Technology
OO	Object Oriented
OOA	Object Oriented Analysis
OOD	Object Oriented Design
PABX	Private Automatic Branch Exchange
PC	Personal Computer
PD	Participatory Design
QFD	Quality Function Deployment
REALIZE	Requirements Engineering by Animating LISP incorporating Z Extensions
SA	Systems Analysis
SEI	Software Engineering Institute
SHU	Sheffield Hallam University
SSM	Soft Systems Methodology
TAS	TranZit Analyser Subsystem
TDSR	Top-Down Stepwise Refinement
TPM	Type Pattern Matcher
TTE	TranZit Transformation Engine
UCD	User-Centred Design
VDM	Vienna Development Methodology

ViZ	Visualisation in Z
WYSIWYG	What You See Is What You Get
Z	The Z Notation
ZAL	Z Animation in LISP
ZIF	Z Interchange Format