WRITING AND ANIMATING Z SPECIFICATIONS

Simon John Andrews

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

November 1996

ABSTRACT

The work presented in this thesis is concerned with the issues involved in writing and demonstrating formal specifications of information systems written in Z. The use of Z in software development, to enhance productivity and improve software quality, is not without its problems. Whilst the notation itself is highly developed, ways of systematically using Z to create specifications are, by contrast, poorly documented. Also, given that most commissioners of software are not skilled in reading Z, ways of demonstrating the important features of a formal Z specification to a customer are needed if the effective validation of the specification against user requirements is to take place. In this thesis we present a systematic approach, known as OPERATOR, for developing Z specifications and evaluate it against the issues identified for writing formal specifications. We also look at various ways of demonstrating Z specifications. We describe how Z specifications may be animated using Crystal, but go on to present a prototype CASE tool, known as Zappa, that may be used to create and demonstrate faithful animations of Z specifications. The thesis starts with a thorough review of software engineering and of the development and rise of formal methods. The development of the OPERATOR approach is then given along with a review of animation, a description of the Crystal technique, and the development of the CASE tool Zappa. An evaluation of the research against the stated aims is presented and areas where future research is needed are pointed out.

ACKNOWLEDGEMENTS

The author wishes to thank the following organisations and people:

Professor Allan Norcliffe

the author's Director of Studies

for his considerable support, advice, effort and encouragement

Dr Glinn Rodgers

Ian Morrey and

Prof Jawed Siddiqi

the author's second supervisors

for their support and advice

Sam Valentine

the author's external advisor

for his guidance and interest

The N. A. B.

for the initial funding of this project

Professors Allan Norcliffe, Peter McEwan and Ian Draffan

for giving the author many teaching opportunities

Sheffield Hallam University,

formally Sheffield City Polytechnic,

and the friendly and helpful staff who work there,

for providing excellent facilities and a pleasant working environment

Dave Marriott

Benoit Dhers and

Alestair Jack

for their work with the author on Zappa's development

Dr Richard Gibson

for providing a Z font and conversation

Dr Mike Tomlinson

for his support and enthusiasm for formal methods in the MSc EIT course

Val Slynn

for help with *Tables* and *Draw* for *Word for Windows*

Carl Ryan-Edwards

for providing scanning facilities

Tony and Brenda

for their financial support and encouragement

and finally

Shan

for everything

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF APPENDICES

Contained in a Separate Volume

CHAPTER 1: INTRODUCTION

Organic life, we are told, has developed gradually
from the protozoan to the philosopher, and this development,
we are assured, is indubitably an advance.
Unfortunately it is the philosopher, not the protozoan,
who gives us this assurance.
Bertrand Rusell, Mysticism and Logic.

1.1 The Research in Perspective.

The research presented in this thesis is the culmination of work started in 1988 as part of the NAB III Office Systems Project. The Office Systems Project involved teams of researchers from the then departments of Mathematical Sciences, Management Sciences, Applied Social Studies, Communications, and Computer Studies at Sheffield City Polytechnic (now Sheffield Hallam University). The Project was concerned with all aspects of developing new office environments together with issues pertaining to the introduction of office automation. One of the components of the research involved the mathematical specification of office systems, and it was out of this that the work presented here grew.

The specific issue to be addressed in this component of the Project was how could one create a formal specification of a typical office information system in an interactive way with the client or user so that the essential features of the specification could be demonstrated without necessarily having to implement the system first. As well as addressing this specific issue, the research, it was hoped, would also help to ameliorate some of the problems associated with the use of mathematics as a specification language in the software development process.

The uptake of the use of mathematics in software engineering (formal methods) has been disappointingly slow, and the reasons for this are many. In 1989, when the author registered for his PhD, the rationale for the research being proposed was given as follows (quoting directly from the Research Degree Registration document):

"The recent introduction of formal methods into the software development process, to enhance productivity and improve software quality, has brought with it several problems.

As well as the problems associated with changing working practices that formal methods inevitably imply, there are also significant problems stemming from the difficulties associated with the reading and writing of mathematics. To create a formal, i.e. mathematical, specification of any computer-based system, the software engineer must be able to build a mathematical model of what the system has to do. He must therefore be able to write mathematics. The customer, for his part, ideally needs to be able to read mathematics in order to understand the specification of the system he has commissioned, so that he can satisfy himself that the system being developed will behave as it should.

Within this particular context the following two points are relevant.

- *Whilst the mathematical notation in which to write formal specifications is highly developed - Z and VDM are two industry standard languages, for example - ways of systematically using these notations to create a specification are, by contrast, very poorly documented. Methods, such as they are, tend to be acquired by software engineers the hard way - in the field, and are not readily passed on to others. There is thus a potential bottleneck here stemming from the problems associated with writing mathematics.*

- *Given that most customers or commissioners of software systems are not skilled in reading mathematics, structured ways of demonstrating* *important features of a formal specification must become part of the specification process. Something midway between animation or prototyping on the one hand, and an accompanying English commentary on the other, is what is ideally needed if the vital exchange of ideas and discussions is to take place effectively between software engineer and customer.*

Within the problem domain of office automation the research work being proposed therefore aims to address each of these problem areas, and to develop systematic and structured ways of building and demonstrating formal specifications."

The rationale for the research is also captured succinctly by Sam Valentine (ex Logica, now at the University of Brighton) in his letter of support for the research, dated February 20th 1989, in his role as industrial advisor to the project:

"The quality of the specification of computer systems is an important factor in the success or otherwise of the eventual implementation, yet methods for capturing specifications are often unsystematic and notations for engineering them are almost always informal.

Formal notations have been developed, but usage of them in the industrial context has hitherto been slight. One factor hindering their adoption is the lack of an agreed method for translating the perceived needs and informal specification into the formal language. Another is the lack of tool support for those languages, of which animation would be particularly useful as a way of providing rapid feedback to clients of the implications of the formal specification as it is developed."

The original aims for the research, set down in the registration document, were actually given as follows:

- To investigate the issues involved in creating and demonstrating formal specifications of office systems and office objects, and to develop systematic ways of facilitating the creation of formal specifications.

These broad aims have not changed to any large extent. However, as the work has progressed over the years there has been a focusing on the three specific aims given below.

1.2 The Aims of the Research.

The aims of the research work contained in this thesis are as follows:

- To investigate the issues involved in creating and demonstrating formal specifications of information systems.
- To develop a systematic approach to creating formal specifications of such systems.
- To investigate ways of animating such specifications.

The focus has become sharper in that the formal specification language being considered is Z (and not VDM or other languages) and the approach to demonstrating formal specifications has become, in the main, one of animation. Reference to information systems, rather than office systems, has allowed the research to have wider applicability.

1.3 The Research Work Plan.

The plan of research proposed has not been deviated from significantly as the research has progressed.

The work was to commence with the acquisition of background knowledge and skills needed to carry out the research. This was to include:

- Fluency in model-based specification languages, in particular the Z notation.

- Expertise in the use of suitable vehicles for animation such as Prolog, C, Crystal and Kappa-PC.

Whilst acquiring these skills a literature search was to be undertaken along with visits to relevant workshops, colloquia, conferences and courses, in order to obtain information on current experiences and practices in the teaching and use of formal methods. The aim was to concentrate on issues to do with the creation and demonstration of formal specifications, rather than on issues of verification and refinement.

From the findings of these investigations, and using the research skills outlined above, various approaches to creating and demonstrating Z specifications of information systems were then to be examined. These were to be tested on a range of examples such as features of security, library, and banking systems. The aim was to involve individuals such as students and software engineering practitioners not familiar with the research. In this way the first of the aims of the research would be achieved.

The building methods were to be evaluated from the point of view of how easy they were to understand and use, the ease of teaching them, and the quality of the resulting Z specification. The building methods were also to be evaluated as to their efficacy in communicating with the customer or user the essential features of the resulting specification, and how effectively the approaches worked as validation tools.

To achieve the second of the research aims, promising techniques were then to be developed further and refined to produce a systematic approach for creating Z specifications of information systems. A systematic tool-based means of animating Z specifications was to be developed, to achieve the final research aim.

1.4 Research Outcomes.

The research work presented here describes in detail the development of a systematic diagram-based approach to creating Z specifications, known as OPERATOR, and the development of a prototype CASE animator called Zappa.

OPERATOR enables a developer, or student, to construct Z specifications from natural language system requirements by first creating diagrams of the system state and system operations. These diagrams convert systematically into Z but, before this is carried out, they may be used to communicate essential features of the system to a would be client or user. When developer and client are satisfied that system requirements are being captured, the diagrams can then be used in conjunction with the original requirements document to produce the formal specification.

Zappa enables the developer to take a Z specification and, provided the specification is in a form suitable for animation by the tool (specifications produced via OPERATOR usually are), to then systematically engineer a working model of the system that is consistent with and mirrors the specification (a form of executable Z specification). This animation can be used to demonstrate the Z specification to the client or user.

1.5 Implementing the Research Work Plan.

As we have said, there has been little deviation from the original research work plan. However, it is important to note that the underlying ideas and associated research behind the development of OPERATOR and Zappa evolved very much in parallel with one another, and in many ways quite separately. There is the temptation to assume, perhaps, that the technique of systematically creating Z specifications would be developed and perfected before thought was given to how specifications could be animated. This was not how the research evolved.

The starting point of the work was a collection of Z specifications (of security systems, banks, libraries, vending machines, stock and production control systems, Email systems, etc.) produced by the author and others. Systematic ways of arriving at these, and various ways of developing working Z models were looked at in parallel. If anything, the major research effort initially focused on the problems of animation until the author had acquired substantial experience of teaching Z, at all levels, allowing the author to appreciate fully the problems faced by students, and experienced programmers alike, when using the Z notation to develop formal system specifications. Only when it had been decided to develop an animation tool (Zappa), using the expert system shell Kappa-PC, was the problem of creating Z specifications in a systematic way seriously considered. By this time the author had considerable experience of writing and teaching Z and, drawing also on the experience of his supervisors, was able to feed this into his research.

It should be noted at this stage that the teaching duties of the author involved teaching Z (including refinement, implementation, basic proof and animation) to a range of students from HND level, through degree, to Masters level. The teaching in the Masters course was extremely valuable because the students ranged from graduates relatively fresh from their degree courses, to programmers, software engineers and other technical practitioners with considerable working experience in the software, computing and information technology-based industries, as well as those from IT or programming departments within a variety of other commercial organisations. Research ideas, as they fed into (and indeed, back from) the teaching, particularly at Masters level, could therefore be evaluated by students and practitioners alike.

During the period of the research, the author has also supervised numerous first and Masters degree projects and overseen many learning contracts directly involving aspects of the research contained in this thesis. This experience has been invaluable and the opportunity to have been involved with students in the classroom is gratefully acknowledged.

1.6 Overview of the Thesis.

The research work in this thesis is concerned heavily with the process of software engineering and the use of formal methods within that process. Consequently it has been necessary to review, in rather more depth than is usual in theses on computer science, the history of software engineering and the development and rise of formal methods. Chapter 2, therefore, is devoted in its entirety to the history of software engineering and the need for formal methods. Chapter 3 looks in detail at the issues of using formal methods in software development together with the issues surrounding the acquisition of formal methods skills and knowledge. In particular, problems associated with learning and using the Z specification language are aired and the clear need for a step by step approach to creating Z specifications is demonstrated along with the need for animation and rapid prototyping tools.

In Chapter 4 the development of the OPERATOR method is considered. The rationale for the approach is first argued and set down, and then the development of the method is traced. We look at how the method has evolved from its origins as a systematic but abstract method developed some 2 years ago, to the fuller method it is today with its graphical front end and its structuring mechanisms for handling system complexity. The results of using the approach in the classroom are included in the chapter.

In Chapter 5 we investigate the status of computer-based tools that support the use of Z and thus point to the need for the provision of animation tools. We consider the issues surrounding the animation of formal specifications and review some of the pioneering work done in this area and highlight fundamental differences in approach. We also look specifically at how working Z models may be created using Prolog.

Chapter 6 is a technical chapter, describing the next phase of the animation research where the opportunities afforded by using an expert system shell for animating Z specifications are considered. Much time and effort was invested in developing a systematic way of animating Z specifications in Crystal - an expert system shell produced by Intelligent Environments Ltd. of Richmond. This research is presented, as well as classroom experience of the approach. Finally we give the reasons for turning to the more sophisticated knowledge-based expert system shell, Kappa PC.

Chapter 7 is another technical chapter where the development of the CASE animator, Zappa, is described and the advantages of Kappa PC over Crystal are explained. The fundamental differences between the two approaches are discussed. The development of Zappa is likely to be ongoing and what is presented here is therefore a description of a prototype. The rationale for developing the tool in the way chosen is given and the use of the tool to animate example specifications is evaluated. Work in developing the tool with students is included.

In Chapters 8 and 9 we critically review the outcomes of the research in the light of the aims of the research, and make our conclusions.

Chapter 10 is the final chapter and suggests areas for future research and development.

CHAPTER 2 : THE DEVELOPMENT OF SOFTWARE ENGINEERING

spatchcock n., & v. t. **1.** n. fowl killed and then plucked, dressed, split open, and cooked immediately.
O.E.D.

2.1 Early Days

If we take the term software engineering to mean simply the production, by whatever means, of computer software, or even just written code, then, arguably, the world's first software engineer was Ada Augusta, the Countess of Lovelace, an able mathematician, who in the 1830s coded sets of instructions for Charles Babbage's Analytical Engine. Of course, the term software engineer was not in use at the time - it was adopted by NATO in the 1960s to describe the process of writing computer programs - and Augusta's sets of instructions were not actually implemented since the Analytical Engine remained conceptual. Nevertheless, the Analytical Engine is now widely recognised as the first computer as we understand the term today. It had memory, input, output, control and arithmetic units, and the operator of the machine could place instructions in the Engine to undertake and reproduce lengthy mathematical procedures.

It was not until the Second World War, however, that computers, utilising the vacuum tube technology of the day, were first employed for practical purposes: in the United Kingdom for decryption and in the United States of America for gunnery table computation.

As hardware technology progressed from the vacuum tube, through transistors and integrated circuitry, to today's Very Large Scale Integration (whereby hundreds of thousands of transistors are etched onto one small silicon chip), so the methods and languages used to instruct or program computers developed. Initially programs were no more than long lists of binary digits. The difficulty which programmers had writing such code led to the development of interpreters and compilers of decimal code, mnemonic assembly languages and eventually higher level languages such as FORTRAN, COBOL, BASIC, C and so-called Fourth Generation Languages. The emphasis was on developing programming languages that were more natural, more easily comprehended by humans, and which could be translated automatically into the low level understood by computers. This development was facilitated by hardware technologists striving for ever faster computational speed, ever smaller scale and increasingly sophisticated computer architecture.

2.2 The Beginning of Software Engineering

It was during the period *circa* 1959 - 67 that career structures began to emerge for programmers, designers and systems analysts as the power and possible applications of computers started to be realised, and, by the early Seventies, once the minicomputer had been invented, that applications moved away from scientific and specialised areas towards more general and widespread aspects of society and industry. The computer began to be trusted in areas where human safety was a consideration. Today every person in the developed world is dependent, to a lesser or greater extent, on software systems. From gas bills to traffic signalling, from medical treatment to the operation of nuclear power stations, computers have a significant role to play.

Burnham [Bur83] puts it thus:

"The computer is the welfare agency, the police, the tax collection office, the insurance company, the bank, the telephone network, the security force and the credit rating firm quietly cataloguing all our works and days."

If we take hardware technology and programming languages to be two strands of software engineering development, the latter dependent, to some extent, on the former, then the expansion of the domain of computer application is the third.

As the complexity and size of computer software increased, so did the need for quality and reliability. Consequently, two further strands of development can be identified. One strand being methods and practices in software engineering, the other, the dissemination to practitioners and students of the ever widening knowledge base. The five strands now outlined are represented in Fig. 2.1.

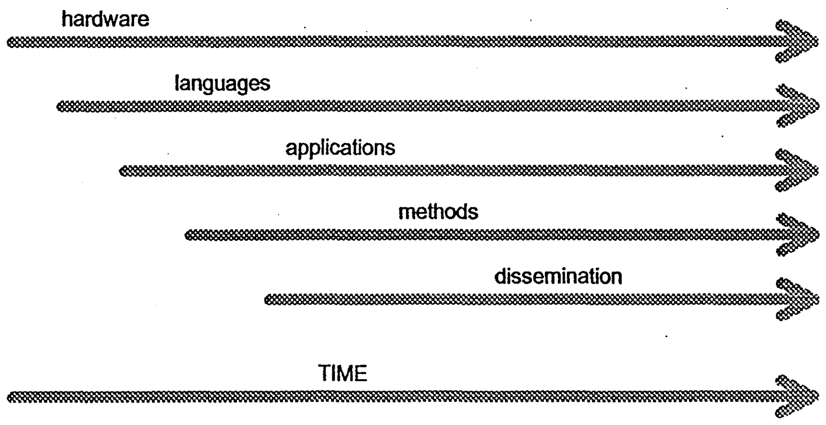


Fig. 2.1: The five strands of the development of software engineering.

In the early Seventies, efforts were made to introduce sound engineering-like practices in the production of software. Top Down Step-Wise Refinement and the use of pseudo code, as an intermediate step between informal system requirements and program code, were expounded (for an example of this approach see Wirth [Wir82]). The first sense of formality emerged in the form of flow charts; still useful today and used later in this chapter to illustrate the software development life cycle. The move was to be away from software production as an art form, practised by software crafters, towards a more scientific, engineering-based, discipline: software engineering as a science in its own right.

In 1972, Bauer [Bau72] defined software engineering as :

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

The definition centres on the need for the use of sound engineering principles and it would be wise for us to attempt to identify some of these principles. We believe (see also Buxton and Marco[Bux87]) that they should include:

- **Good project management** - decision making and administration.
- **A well defined project structure** - a procedure to follow, a modular approach.
- **A common language or languages** - standard notations, diagrammatic representations and documentation in general.
- **Tried and tested methods and techniques** - a wealth of experience and knowledge from which to draw.
- **Scientific foundations** - facts, rules and models.
- **The use of engineering tools**

Software engineering is, then, a very young discipline. It would be a lot to expect it to compare well with mature, highly developed, fields of engineering such as civil engineering, bridge building, aircraft construction, car manufacturing and the like, all of them solidly founded in the sciences of physics and chemistry, and all of them heavily dependent on the use of mathematics.

Indeed, in 1975, Tony Hoare [Hoa75] observed that:
"The attempt to build a discipline of 'software engineering' on such shoddy foundations must surely be doomed, like trying to base chemical engineering on phlogiston theory, or astronomy on the assumptions of a flat earth."

and asked:

"How many of them [software engineers] *are ignorant of, or prefer to ignore the known techniques used by others, and embark on some spatchcocked implementation of their own defective inventions."*

This sentiment had been expressed two years earlier by a clearly exasperated high-ranking United States Air Force decision maker [USA73]:

"You software guys are too much like the weavers in the story about the emperor and his new clothes. When I go out to check on a software development, the answers I get sound like 'we're fantastically busy weaving this magic cloth. Just wait and it'll look terrific'. But there are too many people I know who have come out wearing a bunch of expensive rags or nothing at all!"

(It may be interesting that Tony Hoare wrote a lecture paper in 1981 [Hoa81] describing his involvement, in the Sixties, with the less than successful Elliot 503 Mark II operating system and equally less than useful ALGOL 68 programming language, and entitled it: *"The Emperor's Old Clothes"*, ending it with his own version of the age old tale!)

Another colourful analogy came from Fred Brooke, in 1982 [Bro82]:
"No scene from prehistory is quite so vivid as that of the mortal struggles of great beasts in the tar pits ... Large-System programming has over the last decade become such a tar pit, and many great beasts have thrashed violently in it. Large and small, massive or wiry, team after team have become entangled in the tar. No one thing seems to cause the difficulty - any particular paw can be pulled away. But the accumulation of simultaneous and

interacting factors brings slower and slower motion. Everyone seems to be surprised by the stickiness of the problem."

This was notwithstanding the fact that a software development life cycle had, by this time, been well, if variously, described. Division of labour was now common practice within software houses - we have already mentioned the appearance of distinct professions - and good communication between software developers was important, see Fig. 2.2. Software engineering was now described by several phases, typically; requirements analysis, system specification, design, implementation, testing and maintenance. There are numerous texts on this subject, Sommerfield [Som92] for example, so we give only a resume:

Requirements analysis (or systems analysis) is the process by which the detailed requirements of a system are formulated. The exact nature of this process depends on whether we are designing a system for our own needs or a client's, and whether we are replacing or enhancing an existing system, or are creating an entirely new system. Communication between client and analyst is at a premium at this initial stage.

System specification is the process by which the requirements of the system are translated into a precise specification. A specification document - the software blueprint - should be unambiguous and free from the clutter of design and implementation details. It should make absolutely clear what has to be done but without the constraint of explicitly describing how it should be done: abstraction is the

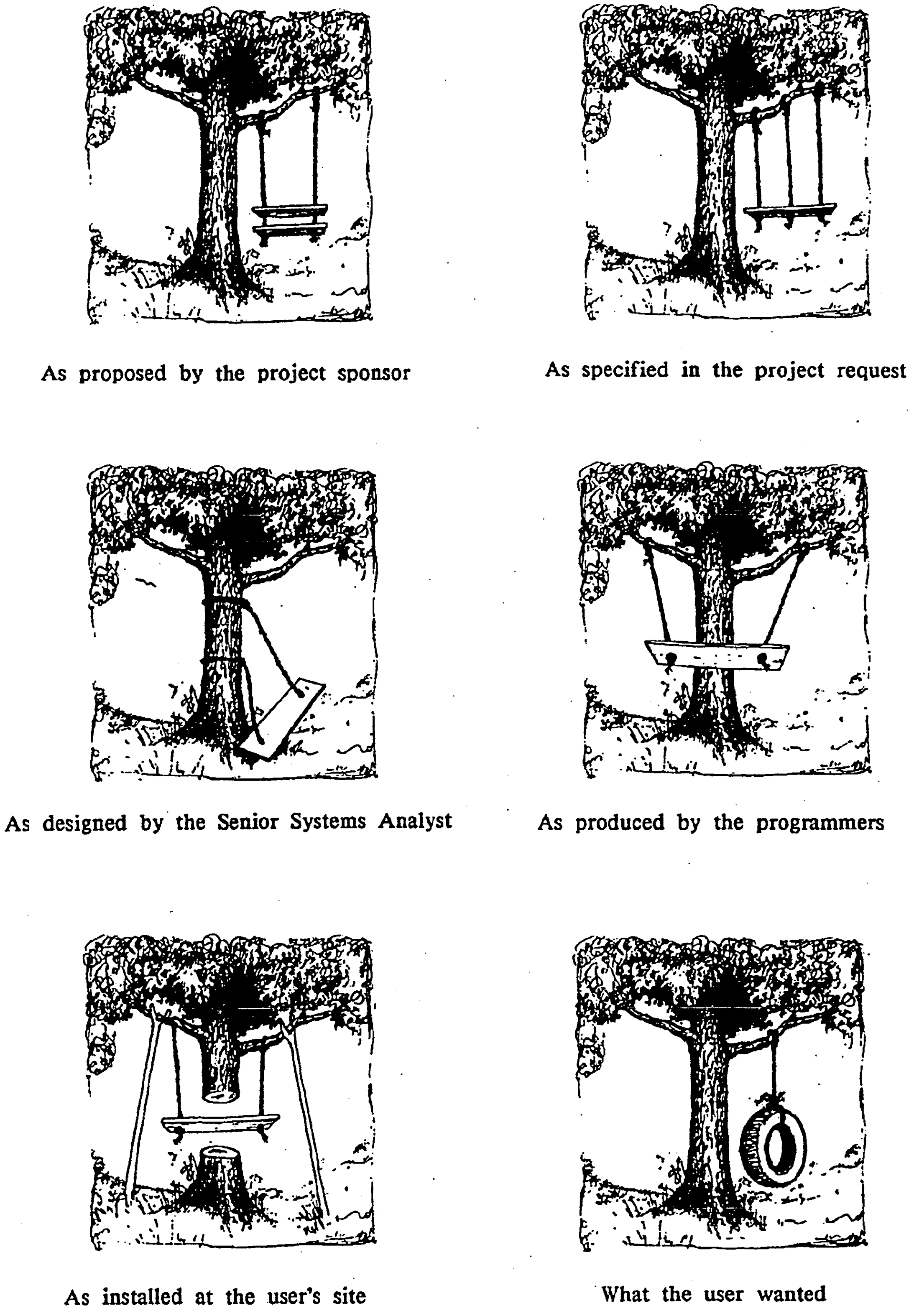

As proposed by the project sponsor
As specified in the project request
As designed by the Senior Systems Analyst
As produced by the programmers
As installed at the user's site
What the user wanted

Fig 2.2: A Problem of Communication.
Source: Essential Mathematics for Software Enginers, Peter Peregrinus, 1987.

key here. Ideally, the specification document should also provide an effective means of communicating the intentions of the software engineer to the client and the systems designer. At this stage the customer should be able to validate the specification with respect to his or her own intentions. A valid specification can be achieved by an iterative process of modification, if needs be.

Design: once the functionality of the system has been agreed upon, the systems designer must specify how it can be achieved. Suitable data structures and algorithms are designed. Consideration of performance and efficiency now comes into play. Decisions regarding hardware and target programming language requirements may be firmed up at this stage. The designer should have a clear and unambiguous system specification document to work from, and should be able to communicate effectively with the originator of the document in order to achieve a correct design that is close to being code.

Implementation is the process by which the data structures and algorithms in the design are coded in the target programming language and installed on the chosen hardware. The emphasis is on correctness of the code with respect to the design, and, hence, to the original system specification. Ideally, a well-defined step-by-step method of translation should be employed, which guarantees the preservation of constraints and conditions laid down in the specification. The more automatic and formal the process, the greater the confidence will be in the correctness of the implemented program.

Testing involves verification of the correctness of the implemented program with respect to the system specification document. Testing is also required to discover and eliminate syntactical errors that may have occurred during coding. Without formal methods of proof or an unusually high level of confidence in the correctness of the program, and especially when dealing with large and complex systems, testing can be very time consuming and, hence, very expensive. This can be true no matter how carefully test data are chosen. It should be noted that testing can only demonstrate that the program is correct with respect to the system specification: if the specification is invalid with respect to the actual requirements then the resulting program is unlikely to implement these requirements.

Maintenance of a system begins on the day that the system first becomes operational and is usually required, on and off, throughout the lifetime of the system. Maintenance may mean modification of the system due to a change in requirements, in which case a clear and unambiguous specification can be of great benefit. Maintenance can also mean the occasional and regular "spring clean" of system data, especially of long lasting data. Human error during data transcription will always be a problem, so stored data should occasionally be verified with respect to original source documents. Unfortunately, maintenance is also, all too often, a euphemism for an ongoing process of debugging.

2.3 Methods of Software Engineering

Fig. 2.3 is a representation of the software development life cycle as outlined in the previous section.

Having a well defined software development life cycle introduced two sound engineering principles; it gave a project structure and a framework for good project management.

As concepts, these stages are clear as to what should be done but not clear as to how it should be done. Methods were needed, so came the advent of Structured Methods which fleshed out the individual stages with procedural details, explicit inputs and outputs to and from stages and ways of checking consistency between and correctness of the products of stages. A key element in all such "semi-formal" methods was the use of various diagrammatic and tabular notations and, consequently, the employment of standard and universal languages to describe system procedures, data structures, functionality, etc. In short, the adoption of the third of our sound engineering principles, that of common language.

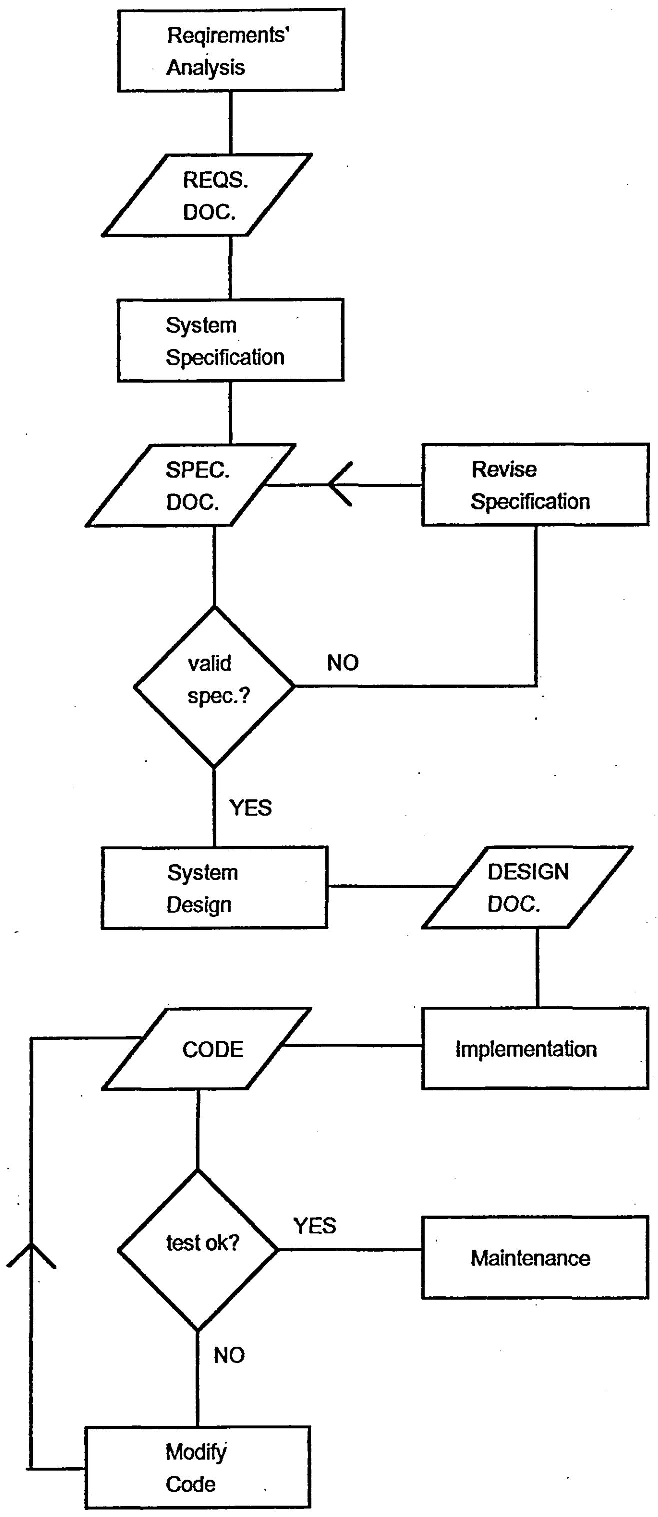


Fig. 2.3: A flow chart illustrating a software development life-cycle.

An early example (early to mid Eighties) was Jackson Structured Programming (JSP) [Jac85]. This method bases program design on the characteristics of the data to be processed by the system. A well defined diagrammatic notation is used to represent hierarchical data structures, illustrating, progressively, more detail through a process of data refinement. A preliminary version of the program structure is then produced by identifying processes acting on the data structure. Once activities and conditional elements have been detailed within processes a detailed program design is derived that *'can be converted directly into programming language statements.'* [Bur87] The same semi-formal diagrammatic notation is used throughout and the diagrammatic output from one stage is the input to the next stage, in which it is either enhanced, modified or refined in some way. However, JSP is not a complete method, in that it does not embody all of the software life cycle; it does not concern itself with requirements' analysis or the formation of a specification document (a specification document constitutes the start point of the method), and many of the stages in JSP rely on manual examination of the specification. There can be no formal guarantee of the validity of the resulting program.

In the mid Eighties, the United Kingdom government's Central Computer and Telecommunications Agency decided they needed a method of software development with the following features:

- A self checking mechanism
- Tried and tested methods
- Facility for tailoring
- 'Teachability'

One Structured Method came to the fore: the Structured Systems Analysis and Design Method (SSADM) [Dow92, Ham93]. SSADM became the United Kingdom government's standard method for carrying out the systems analysis and design stages of information system projects and is now a recognised international industry standard. Indeed, in many application areas, in particular in safety critical system development, it is a legal requirement to use SSADM. The Ministry of Defence in the United Kingdom, for example, must use SSADM for all of its software development projects. Its popularity and widespread use has meant that it has undergone frequent revision and enhancement, its fourth incarnation, SSADM version IV, was released in July 1990.

SSADM takes the structure for software development and project management forward. The software development life cycle is defined by five core stages or *modules*, roughly corresponding to the stages outlined previously, but now each *module* has within it one or more *stages*. Each *stage* is further broken down into *steps* and these, in turn, into *tasks* within each *step*. It is made explicit, at each level, as to when something is to be done. Each *task*, *step*, *stage* and *module* gives rise to *products*; usually in the form of well defined documents. Particular *products* form the material required to accomplish the next *task*, *step*, *stage* or *module*. SSADM also supplies a raft of *techniques* to be applied within *tasks* in order to obtain the appropriate *product*. SSADM, then, describes **what** should be done (*products*), **when** things should be done (*modules*, *stages*, *steps* and *tasks*) and **how** things should be done (*techniques*). These aspects are represented in Figure 2.4.

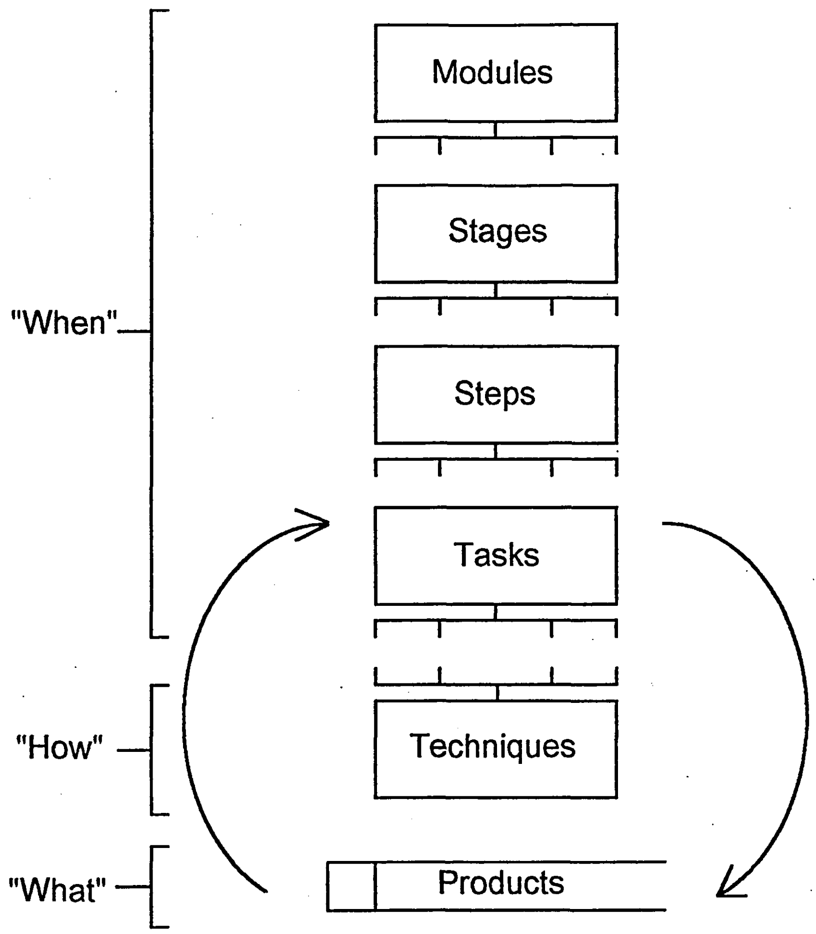


Fig. 2.4: Aspects of SSADM.
Source: Downs, Clare and Coe, 'Structured Systems Analysis and Design Method: Application and Context'.

It is not our purpose to describe the large number of SSADM *tasks* and *steps*, here (there are over 30 *steps* and around 150 *tasks* in the latest version), but a list of the *modules* will at least allow comparison with the software development life cycle described earlier.

The five *modules* of SSADM are:

1. Feasibility Study
2. Requirements Analysis
3. Requirements Specification
4. Logical System Specification
5. Physical Design

It is notable that there are clear levels of abstraction and the process is one of refinement from the most abstract model, given by the Requirements Specification *module*, which is a *'detailed and testable'* specification of what is required, to a non-procedural, logical design (given by the Logical System Specification *module*) which is independent of any implementation strategy, and finally to a physical design which introduces information about the target hardware, software and the organisational setting in which the system will operate. The basic software life cycle, however, remains the same.

The *products* of the various *tasks* are classified as either *structural model diagrams*, *supporting text* or *reference text*. The structural model diagrams form the core of the system description. They take the form of tried and tested semi-formal diagrammatic notations such as Data Flow Diagrams, Entity Relationship Diagrams, structure charts to model entity life histories, graphical models of entity attribute relationships, matrices to cross reference such things as entities and data stores, and process outline diagrams. Many structural model diagrams are inter-related and many begin as a logical model to be refined to a physical model. Much of the format and content of the supporting and reference text is standardised using a variety of proformas.

As an industry standard SSADM enjoys excellent textual, tool and service support. The 1990 SSADM Directory of Services [CCT90] listed 139 organisations supplying consultancy, 28 accredited training agencies, 30 suppliers of Computer Aided Software Engineering (CASE) tools and 35 suppliers of fourth-generation languages who have published interface guides for SSADM compatibility. SSADM is widely taught in further and higher education establishments.

By 1992 SSADM was being used on billions of pounds worth of software development in the United Kingdom alone [Dow92].

However, whilst SSADM is designed to be flexible and can be "part used", smaller organisations may find the extra effort involved in using such methods unattractive. The time and costs involved in a radical change in working practice, such as staff training, tools and services, may be disincentives. The quantity of documentation that results can be difficult to manage and, combined with the large number of procedures involved, there might appear to be a need for a significant increase in project administration.

To some extent these problems have been addressed by the development of more affordable tools, often PC-based.

2.4 Computer Aided Software Enginering

In general, CASE is an attempt to speed up, help manage and simplify many procedures within software development, particularly in the use of structured methods. Any form of automation is usually perceived as a "good thing". Fairburn [Fai90] puts it thus:

"It is the rule rather than the exception that automating processes improves the quality, consistency and reliability of products over any but the most

meticulously and expensively hand-crafted products. It would be surprising if this were not true also in the field of software production."

In any case, the ever increasing size and complexity of software engineering projects makes it obvious that some degree of automation in production would be desirable if not, for the largest of systems, essential:

"How do you write the specification for a new project that has to meet 2,000 main performance requirements and 6,000 detailed requirements without any of them conflicting with each other or overlapping? How do you weed out ambiguity and over-engineering? How do you keep track of changes in the specification and the decisions behind each one?"

asks John Dunn in a recent edition of the *Production Engineer* [Dun94]. He is referring to the Civil Aviation Authority's 350 million pound national en-route traffic control centre, being built at Swanwick, Southampton, a highly safety critical system, heavily dependent on large and complex software systems. The answer, apparently, was to use a computer program called Requirements and Traceability Management (RTM), developed by GEC-Marconi in the mid to late Eighties to aid in the development of complex military systems. A sort of automatic project manager cum validation aid, RTM is a text-based database program in which information from specification documents is "captured, tagged and sorted". It attempts to identify everything the product is required to do and present its findings in a way the client can understand and validate the specification. In the case of the national en-route air traffic control centre, 15,000 initial requirements were reduced to 2,000.

CASE tools appear to fall into three loose and overlapping categories; those that are designed to handle a particular procedure in a software engineering method, those that are designed to oversee a complete project from beginning to end (or at least a significant portion of a project) and those that are targeted at a particular programming language or type of project.

In terms of automation in structured methods (in particular SSADM and derivatives), Parkinson [Par91] describes tools as being *single technique*, *single phase* or *multiple phase*. According to Parkinson the evolution of CASE began in 1978-1983 with first generation tools such as simple 'diagrammers' and project management aids. Second generation tools (1984-1986) included multi-diagrammers that were able to work with two or more types of diagram and sometimes form links between them, software dictionaries to store, collate and manage things like variable and procedure details, and simple 'rapid prototypers'. The third generation of tools (1986-1990) had added intelligence in diagram management, generation and manipulation, and included the first code generating systems (beginning to realise the software engineer's dream (or nightmare) of programs that write programs). In Parkinson's fourth generation of CASE tools (1991-1993) the industry saw the rise of dedicated support for structured methods and so-called Integrated Project Support Environments (IPSEs) "developed from the need to manage complex, large scale software projects, usually in telecommunications, aerospace and defence industries (RTM, although in use well before 1991, is a good example).

IPSEs are tools that fall into our second loose category; tools that attempt to address quality control and management by automating aspects of requirements' capture, design, construction and testing stages of a software development life cycle.

A useful 'mini-catalogue' of commercially available CASE tools can be found in Fisher [Fis88]: *Teamwork* from Cadre Technologies, *Excelerator* from Index Technology and *PowerTools* from Iconix Software Engineering are three examples given of tools that support structured methods, integrating tasks that involve such things as Data Flow Diagrams, structure charts, Entity Relationship Diagrams and data dictionaries. Also included are tools that would fall into our third loose category such as *BRACKETS* from Optima which generates COBOL procedures and data structures, and *TAGS/IORL* from Teledyne Brown Engineering, designed to aid in the production of ADA programs. (A more complete and wide ranging collection of software engineering tools and methods can be found in the STARTS Guide: Standard Descriptions [STA87], which lists most tools and methods currently in use or under development.)

2.5 The 'Software Crisis'

Software engineering, then, has evolved rapidly in the last twenty or so years (and we have not had the space here to include innovations in software metrics and quality assurance methods , the development of Object Oriented Programming and Design, and a host of other developments). All but one of our six principles of sound engineering practice would appear to have been addressed; project management, project structure, common languages, methods and techniques, and engineering tools. Indeed, Bauer's 1972 definition of software engineering now seems to lack detail, given the current software engineering environment, and Buxton and Marco [Bux87] have proposed an up-to-date version:

> *"The establishment and use of sound engineering principles and good management practice, and the evolution of applicable tools and methods and their use as appropriate, in order to obtain - within known and adequate resource provisions - software that is of high quality in an explicitly defined sense."*

(It is possibly less than encouraging that the above quote came from a text entitled *The Craft of Software Engineering*.)

Notwithstanding all of the above it is still widely argued that the industry is in a state of (almost perpetual) difficulty; the 'software crisis'. Potter, Sinclair and Till [Pot91] put it thus:

"The accumulated public perception of computer systems is that they are inherently faulty. Errors are casually referred to as 'bugs', whereas the equivalent term in other engineering disciplines is 'faulty component'. This has led to a rather sloppy situation, where low standards are generally accepted as normal."

A recent BBC Radio 4, *File on Four*, programme [BBC93] was devoted to the 'software crisis' and contained some startling (and worrying) evidence:

On an August weekend in 1993, at the Stockholm Air Show, a Swedish Air Force Saab Grippon Fighter jet was going through its paces. The aircraft was in a 3000 foot climb and banking when the pilot lifted the nose a further four or five degrees. It stalled and crashed. Luckily, the pilot was able to eject and there were no casualties. The reason the aircraft crashed was that the software on board that was supposed to make fatal manoeuvres impossible was not complete. According to Saab, the pilot had attempted something that was "highly unlikely". It was found that Saab's test routine for the Grippon had been "undemanding".

The Royal Air Force have also suffered from shortcomings in its own software: in April 1992 a Harrier Jump Jet, on loan to the Royal Navy (and therefore operating outside of its normal environment) was on a practice run when it dropped a bomb on its own carrier, the Ark Royal. The target had been a dummy, towed some 800 meters behind the ship. The target acquisition system on board the Harrier had been unable to process incoming data fast enough resulting in the bomb tracking the aft flight deck of the ship. The bomb impacted on the deck and penetrated several levels injuring several sailors. The pilot of the Harrier was mortified. He had been "let down by his software".

On October 20th, 1992, the London Ambulance Service switched on their new command and control computer system. Two days later it had to be switched off after delays in ambulance arrival times of up to ten hours had been reported. The Department of Health commissioned a report into the failure which found that "just about everything that could go wrong had gone wrong. The software was not complete, had not been fully tested and still had dangerous mistakes in it."

Speaking on the same *File on Four* programme Cliff Jones pointed out:

"Taking a program of 10,000 lines of code ... there are more paths through that program than there have been seconds in the existence of the universe."

In 1993 British Nuclear Electric was installing a software system for the primary protection of its Sizewell B nuclear power station. It has 100,000 lines of code. They cannot say how safe it is.

In September 1991 there was a narrow escape at British Nuclear Fuels' reprocessing plant at Sellafield, where highly radioactive waste is solidified into glass at the vitrification plant. A crane hoists containers containing the waste into a cell where the process takes place. Two doors are supposed to prevent human access whenever containers are in the cell. Protection software, designed to prevent both doors being open at the same time, was deliberately modified by technicians, pressed for time, to circumvent the safety measure at a time when no danger was present. They forgot to undo their 'hacking' and later a container was hoisted into the cell with both doors open. In was found that a second level of software protection, that would have prevented the incident, was faulty. The fault was a plus sign missing from one line of code.

Martin Thomas, again speaking on the *File on Four* programme, thinks more professionalism in software engineering is needed:

> *"I characterise the industry really as a craft industry that ought to be an engineering industry. We are a very young industry; we've only been developing software in industry for 30 or 40 years, at the outside, and yet we are tackling very large scale engineering problems."*

It is not only in safety critical areas that good software is important. Information systems of all kinds play a crucial role in the running of governments, economies and innumerable aspects of our daily lives. Surely it is at least desirable for there to be a degree of confidence in *all* the software systems that we use. We may be mildly amused by stories of computers erroneously sending individuals million pound electricity bills, but it would be less amusing if the recipient was a person with a heart condition who suffered an attack as a consequence, or if millions of individuals were sent erroneous bills.

Bell, Morrey and Pugh [Bel92] cite "one of the few pieces of hard evidence available" of the 'software crisis', that of a study of a 1984 United States Federal software projects that found that less than two percent of software was used as delivered, out of a budget of 6.2 million dollars (see Figure 2.5).

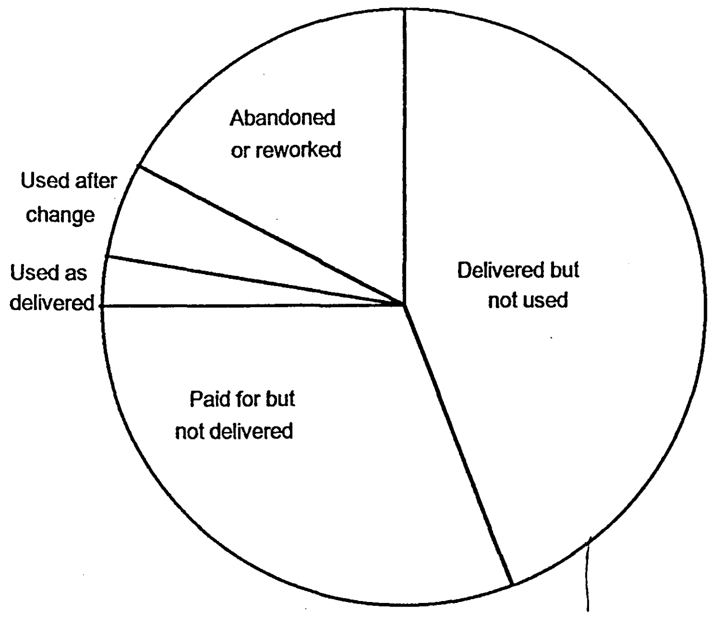


Fig. 2.5: Effectiveness of US Federal software projects carried out in 1984
Source: Bell, Morrrey and Pugh, 'Software Engineering: A Programming Approach',1992.

So it appears that something is missing from the "craft" of software engineering; a solid and scientific foundation from which to build. As *File on Four* put it:

"The computer industry has progressed so fast that it is taking on tasks for which it still lacks the basic analytical tools. It's like trying to build bridges without having first invented the set square."

Traditional engineering disciplines draw heavily on mathematics. Perhaps the use of mathematics in software development is what is missing.

REFERENCES IN CHAPTER 2

Bur83. Burnham, D., *The Rise of the Computer State*, Redwood Burn, 1983.

Wir82. Wirth, N., *Program Development by Stepwise Refinement*, Comms. of the Association of Computing Machinery, 1971, reproduced in *Writings of the Revolution: Selected Readings on Software Engineering*, Yourdon, E., (Ed.), Yourdon Press, N.Y., 1982.

Bau72. Bauer, F.L., *Software Engineering*, Amsterdam: North Holland, 1972.

Bux87. Buxton, J. and Marco, A., *The Craft of Software Engineering*, Addison-Wesley, 1987.

Hoa75. Hoare, T., quote in lecture notes of Norcliffe, A., Sheffield Hallam University, quote dated1975.

USA73. High-ranking USAF decision maker, quote in lecture notes of Norcliffe, A., Sheffield Hallam University, quote dated 1973.

Hoa81. Hoare, C.A.R., *The Emperors Old Clothes*, Comms. of the Association of Computing Machinery, reproduced in *Writings of the Revolution: Selected Readings on Software Engineering*, Yourdon, E., (Ed.), 1982, Yourdon Press, N.Y., 1981.

Bro82. Brooks, F., *The Mythical Man Month*, Addison-Wesley,1982.

Som92. Sommerfield, I., *Software Engineering*, 4th Edition, Addison-Wesley, 1992.

Jac85. Jackson, M., A., *Principles of Program Design*, Academic Press, 1985.

Bur87. Burgess, R., S., *Structured Program Design using JSP*, Hutchinson, 1987.

Dow92. Downs, E., Clare, P. and Coe, I., *Structured Systems Analysis and Design Method: Application and Context*, 2nd Ed., Prentice Hall, 1992.

Ham93. Hammer, M.J., *SSADM Version 4: Project Manager's Handbook*, McGraw-Hill, 1993.

CCT90. Central Computer and Telecommunications Agency, *SSADM Directory of Services*, London: CCTA/BCS, 1990.

Fai90. Fairburn, D., *The Opportunity of the Decade*, paper in *CASE on Trial*, Edited by Spurr and Layzell, John Wiley and Sons Ltd., 1990.

Dun94. Dunn, J., *Give them what they asked for*, lead article in *The Production Engineer*, March 24th, 1994.

Par91. Parkinson, J., *Making CASE Work*, NCC Blackwell Ltd., 1991.

Fis88. Fisher, A.S., *CASE: Using Software Development Tools*, John Wiley and Sons, 1988.

STA87. The STARTS Guide (Standard Descriptions), 2nd Edition, Volume 2, NCC Publications, 1987.

Pot91. Potter, B., Sinclair, J. and Till, D., *An Introduction to Formal Specification and Z*, Prentice Hall International (UK) Ltd., 1991.

BBC93. BBC Radio 4, *File on Four* on *Software Engineering*, Oct. 19th, 1993.

Bel92. Bell, D., Morrey, I. and Pugh, J., *Software Engineering: A Programming Approach*, 2nd Edition, Prentice Hall International (UK) Ltd., 1992.

CHAPTER 3: FORMAL METHODS

The writing will be spare and lean,
the concepts hard, the philosophy old
and yet new born.
John Steinbeck on *East of Eden*.

3.1 Benefits of Mathematics

It may not be immediately apparent how mathematics can play a major role in software engineering. Just because mathematics is an essential modelling and design tool in more traditional engineering disciplines, why should it be equally essential in software engineering? However, mathematics does have certain properties that would make it well suited to the specification of software:

- **Precision**: mathematics is a more restricted language than, say, natural language, and therefore allows fewer opportunities for ambiguity. A mathematical statement is less likely to be misinterpreted.

- **Brevity**: a large amount of information can be conveyed concisely in a mathematical language. Brevity can often be an aid to understanding.

- **Expressiveness**: mathematics is a powerful language in which to express complex ideas.

- **Facility for reasoning**: mathematical statements can be formally investigated. Mathematics opens the way for calculations to be performed for predictive purposes. Ideas expressed in mathematical terms lend themselves to the formal, mathematical process of proof.

In addition to the above, mathematical descriptions can be very abstract. This can be of great benefit during the early stages of system development allowing the software engineer to concentrate on the functional aspects of the system without the clutter of implementation details. In other words, the engineer can concentrate on *what* the system is to do without worrying about *how* this is going to be achieved. Valentine [Val87] cites this as a clear advantage over the use of algorithmic languages for system specification. An abstract mathematical specification says less than a definite algorithm and should therefore be easier to write. He points out that aspects of a specification which, at an early stage, need not concern the software engineer can, initially, be left out of a mathematical specification. Such things could include interface designs, hardware details and the target language for coding. These details and such things as performance requirements can be added later in the software development process.

Wordsworth [Wor92] illustrates this idea of abstraction with two versions of a simple set of system requirements:

"I need a system that will accept certain information as input and produce certain other information as output according to a certain rule."

Compared with:

"I need a system that will accept a text file marked up with certain special signs, and produce formatted pages on a laser printer, the correspondence between the input and the output being given by the following rule..."

The first set of requirements may be so abstract as to be practically useless as a starting point for specification (at least without some idea of the rule to be used) and the second set could be made more specific, or *concrete*, by stating the type of text file (ASCII or WordPerfect, for example), by describing the special signs or by giving details of the formatting required. From this one can begin to formulate a notion of *refinement* of a very abstract specification through specifications that are consecutively more and more concrete until a final, complete and concrete design is achieved - as a painter may begin with sketches, then an outline drawing, before adding washes, detail, solid colours and final touches.

A mathematical description may be declarative rather than procedural, in terms of its being a logical statement that can be shown to be either true or false. Hoare [Hoa86] sees this offering a further advantage over program code when checking correctness. If a mathematical proof of a specification can be given then it should be easier to spot a flaw:

> *"This is because a proof checker only needs to check the validity of each line of proof, comparing it only with one or two previous lines. For a program, the checker has to check each line in the context of every other line of code in the program - a task which is quite impossible for large programs."*

A specification is the key software engineering document; all subsequent processes and stages of system development are dependent on or involve referring to such a document. Ince [Inc88] notes this and suggests a set of important properties that a specification should posses. He proposes that a specification should:

- *be unambiguous*, since a wrong interpretation only detected in the final stages of development could be costly to resolve and even disastrous if only discovered after implementation;

- *be free of design and implementation directives*, allowing an unfettered approach to problem solving and leaving as much choice as possible in terms of algorithm design;

- *enable the developer to reason about the properties of the system it describes*, which lies at the heart of the analysis process that gives rise to a valid specification;

- *be free of extraneous detail*, thus containing no more information than is required for the person who is to process the document;

- *be partitioned* into smaller parts or modules which are, as far as is possible, independent of each other, allowing consideration and modification of one part with minimal consideration of or effect on others;

- *be understandable* by the customer of the software system so as to improve the chances of achieving a valid system.

So far in our discussion it would appear that a mathematical specification could possess all but the last two of these properties: partitioning and 'understandability'. However, there is no reason why a mathematical specification cannot be modular, so we are left with the problem of 'understandability'. Clearly, it would be unfair to expect all software system customers to be mathematically literate.

Mathematical methods (or formal methods as they are now known, thus making them clearly distinct from Structured Methods, that may be thought of as semi-formal methods, and from informal methods, such as the use of pseudo code and Top Down, Stepwise Refinement) may have other benefits or "spin-offs" as Norcliffe describes them [Nor91], to system designers, programmers and industry in general. Given an unambiguous formal specification, designers should know exactly what to do. The process of coding can become more formalised and mechanical, with each piece of code then able to be checked systematically to see that it meets its specification, thus building an inherent correctness into the resulting program. Theoretically, verifying that a program fulfils a formal specification could be done by machines using program provers. In fact, mathematics, with its formality, rigour and logical nature, should lend itself readily to investigation and manipulation by computer based tools (we investigate CASE for formal methods in a later section in this chapter). The process of mathematical specification itself, a form of mathematical modelling, strongly encourages deep thinking about the system being specified which inevitably results in better communication between software engineer and client, and consequently a clearer and faster validation process. A course entitled *Essential Mathematics for Software Engineers* [Sla87] has been produced by a consortium of Sheffield City Polytechnic, The Hatfield Polytechnic, Loughborough University of Technology and ICL Software Engineering. It justifies the word *essential* by stressing that good communication is a prerequisite for effective software development and that good communication is far more likely if a precise language such as mathematics is used:

> *"... we must have a language which allows, and encourages, precise and logical thought and expression. We must have a language which discourages, and prevents, vague and fuzzy thinking and expression. The language of English has developed over the centuries to allow man to express vague and fuzzy thoughts, such as poetry and politics. The language of mathematics has been developed over an equally long time to fulfil exactly the needs which we require for thinking about programs."*

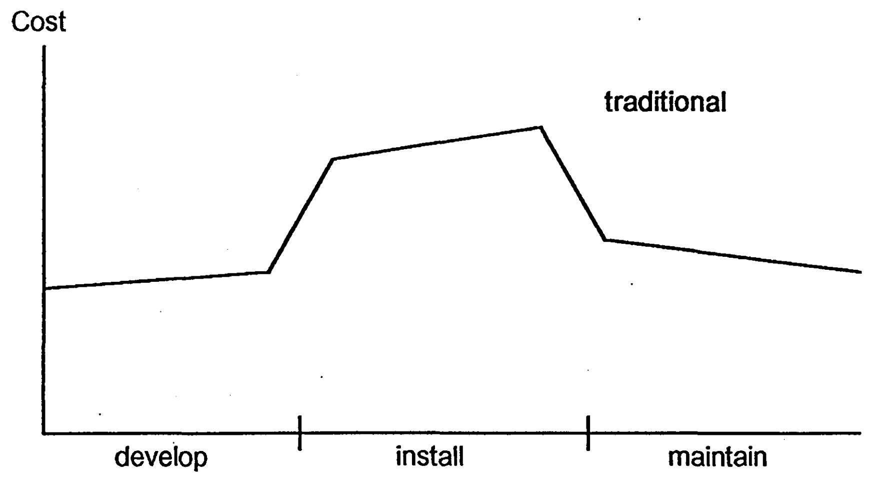
At Sheffield Hallam University, amongst students on the MSc Engineering Information Technolgy, MSc Computer Studies and BSc Computing Mathematics courses, the author has certainly found, during three years of teaching formal methods, that formal specification encourages deep thinking about the system to be specified. As lecturers we are often quizzed about real or imagined intentions within our system requirements documents. Any ambiguities are soon rooted out.

Mathematical specification should also improve communication between software engineering team members involved in various stages of a project if a single, precise language is understood by all. Maintenance of formally specified systems should also become easier with a precise specification document to refer to. If the whole point of formal methods is the production of better software then less maintenance should be required.

Formal methods clearly put more emphasis on the early stages of software development; the production of an initial, abstract mathematical specification is key to the whole process. The extra investment in effort towards *getting things right early on* should mean that significantly less time and effort will be required during later stages, where currently vast resources are used in employing programmers to check and re-check, test and re-test and rewrite and maintain large and complex programs. It is an adage that we must all be familiar with: that preparation is the key to the successful completion of a task. Figure 3.1 illustrates this principle by graphically comparing the resources used at different stages of software development using current methods and formal methods. If the area under each graph represents the total resources used then it is clear that formal methods should be more productive [Sla87].

Many academics are clearly enthusiastic about formal methods. Martin Thomas of the BCS (at the time this comment was made) is certainly in favour [BBC93]:

"I would like a requirement that safety critical software should be fully analysable from its specification right through to its implementation as a computer program, and what I mean by fully analysable is that you should be able to reason formally, using mathematical logic, about the way in which it is



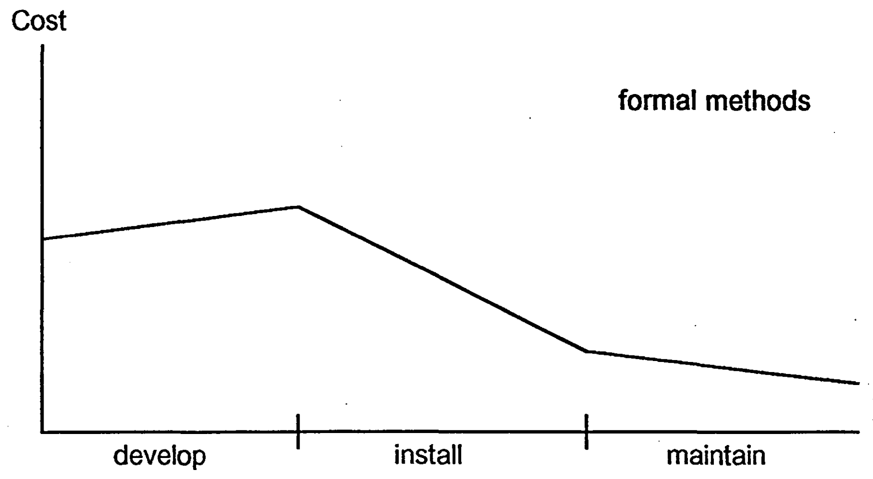


Figure 3.1: Comparison of software engineering cost over project stages between traditional and formal methods.
Source: Essential Mathematics for Software Engineers, Edited by Gill Slater, Peter Peregrinus Ltd. on behalf of the IEE.

possible for that software to behave and the ways it is not possible to behave. So you should be able to say, 'this engine controller cannot lead the engine to overspeed under these circumstances'. And if called upon to do so you should be able to create a mathematical proof that that is correct. Other engineers use mathematically based methods for carrying out their designs and analysing them. We have such methods, some organisations are using them. It seems to me that we need to accelerate the pace of take-up of these methods in industry by actually legislating for it in some way; building it into standards."

If formal methods can enhance productivity and quality then we would concur with the above view but suggest that such improvements need not be restricted to safety critical areas.

Bev Littlewoods could be described as an enthusiast with reservations [BBC93]:

"Mathematics is an idealisation - we don't actually think that way. Computers tend to do things for people, unfortunately ... They [Formal Methods] *are not **the** answer, but they are **an** answer. Certainly they are going to be a contribution to building safer systems."*

The UK Technology Foresight Programme on IT and Electronics hopes that formal methods will be taken up by the software engineering fraternity. In its latest Delphi Questionnaire [For94] it asks:

"When will it be that ... 25% of professional programmers use formal techniques for the design, generation or validation of software?"

3.2 Problems Associated with Taking the Mathematical Approach

The greatest task to be undertaken for there to be widespread adoption of formal methods will be in developing the mix of training needed for professional software engineers to acquire the necessary skills profile and to manage the major changes in working practice that will ensue. There would be more people involved in specification and probably less people concerned with writing and testing software code. Because formality lends itself to automation, redundancies within the industry are implied, as software engineering "catches up with" traditional engineering. The current software engineering work force is largely not mathematically trained and the resources required to retrain such a large and diverse body would be considerable.

Mathematics is perceived, perhaps with some justification, to be difficult and experienced programmers may be sceptical as to its relevance. The resistance to such a radical change in approach has been and will be great. Maibaum and Sadler [Mai28] are well aware of this problem:

"Regarding the practice of formal methods in industry. There is a natural conservatism in all organisations against the use of new techniques and methods. Jobs depend on performance and tried, if not proved, methods are more dependable than new wonder cures. Old ways of working are familiar and dependable, if not so effective. The job will be done, even if it is troublesome and may exceed the budget...It takes courage on the part of a company manager or individual to take up the cudgels of formal methods and risk all."

Djikstra [Dji], however, is uncompromising:
"We have already heard all the objections, which are so traditional they could have been predicted: 'old programs' are good enough, 'new programs' are no better and are too difficult to design in realistic situations, correctness of programs is much less important than correctness of specifications, the 'real world' does not care about proofs, etc. Typically, these objections come from people that don't master the techniques they object to."

Nevertheless, senior management in the software engineering industry need to be convinced that formal methods do indeed lead to greater quality and productivity. In the end this will only come about if commercial software, at least equivalent in quality to software produced using current methods, can be shown to be produced at lower cost, either in time or in resources. Academic institutions can play a leading role in developing and using formal methods and appropriate tools , in training future software engineers and in creating opportunities to collaborate with interested parties in industry. Upon them is the responsibility of dissemination.

Norris, Newsman and James [Nor87] agree:
"... the current situation may be changed by a more mathematical basis to the teaching of computer science or by good support tools. In any case, it will be a while before the status quo *changes significantly, and the type of tools required to solve some of the practical problems of formal methods will be a quantum leap from the average compiler... A likely next step in the evolution of formal methods would be the application of artificial intelligence techniques to prototype such tools."*

Tool support for formal methods is certainly another area of concern, in that they are needed but are currently thin on the ground, at least in commercially available forms. Plat, Katwijk and Toetenel [Pla92] echo this:

"... formal methods need automatic support (tools). Error-free specifications (necessary for reasoning) can be constructed faster when using syntax-directed editors and type-checkers, and non-trivial proofs tend to be too complicated to be carried out depending on human intelligence alone...Nevertheless, the current availability of tools is low, and those tools that are on the market offer a limited form of support."

Gibbins [Gib88] also sees CASE tools playing a vital role:
"It may be that the future applications of formal methods lie both in the development of software tools - syntax-checkers, proof-checkers and theorem provers - which enables control of the formal software development process, and in an associated prototyping methodology which enables one to test formal specifications."

Another problem involves communication difficulties. We have suggested that a formal approach necessitates a deep understanding of a system by the software engineer and that this can only help towards more effective discussion with the software client but discussion is not enough in itself. A client wants to see something concrete and something that he or she can comprehend, and wants to see such things early in the system development process. Michael Jackson [Jac87] sees mathematics imposing difficulties in this area:

"Significant parts of what software developers produce must be discussed, explained, negotiated and eventually agreed with users and customers. These activities must be carried out in the 'domain language', the language that users and customers rely on when they speak of the real world in which they operate. There is therefore an important requirement for translation and interpretation between the formal language and the 'domain language'. It would be foolish and arrogant to castigate our users and customers for their refusal or inability to learn our formal languages, partly because we simply have no right to impose such an obligation on them, and partly because formal languages are unsuited to human communication."

Diagrammatic documentation, such as that produced by structured methods, is much more likely to elucidate the developer's ideas than pages of mathematical text. Techniques of rapid prototyping have evolved to allow clients to interact with systems at an early stage. If we accept that using formal methods means more time and effort will be spent on system specification, and that it is highly desirable to obtain a valid specification before progressing, then it seems essential that some method be available to convey the meaning of formal descriptions to those not familiar with the notations used.

Possibly the most worrying limitation of formal methods (at least to those currently engaged in commercial software production) is the lack of actual methods. A mathematical notation is a powerful modelling tool but on its own cannot be described as a method. McDermid [Der87] feels that in the area of formal methods *"there is too much emphasis on the notation and too little on the methodological aspects of their use."* Michael Jackson [Jac87] is characteristically blunt:

> *"Formal methods tend not to be methods; most formalists are simply not interested in method except in a very attenuated form... Today's formalisms tend to be isolated from one another, research concentrating on improving each formalism in its isolation; we need to build many bridges between different formalisms, converting our existing archipelago into the solid ground on which software developers should be able to stand."*

It is at least now well understood that formal methods embraces formal specification with stages of refinement and verified design through proof (the idea that programs produced using mathematics can be formally reasoned with, with respect to their formal specifications, from Jones [Jon90], for example). Guides to usage such as Fig. 3.2 [Nor87] and various texts such as [Ear86, Hay87, Kin89, Lit92] have suggested strategies and environments for specification, refinement and program design but it is less clear how one reaches a formal specification from informal requirements.

Ian Sommerville [Som92], whilst not purporting to give a detailed methodology, does suggest an overall strategy which incorporates formal specification (see Fig. 3.3). Formal techniques could be incorporated into existing regimes of software development where stages of development are seen as being inter-related and iterative processes. Although Sommerville believes that formal specification techniques are now sufficiently mature for them to be used in the specification of sequential systems he warns that *"we need even better tools, techniques and methods and perhaps most importantly, better education and training"*.

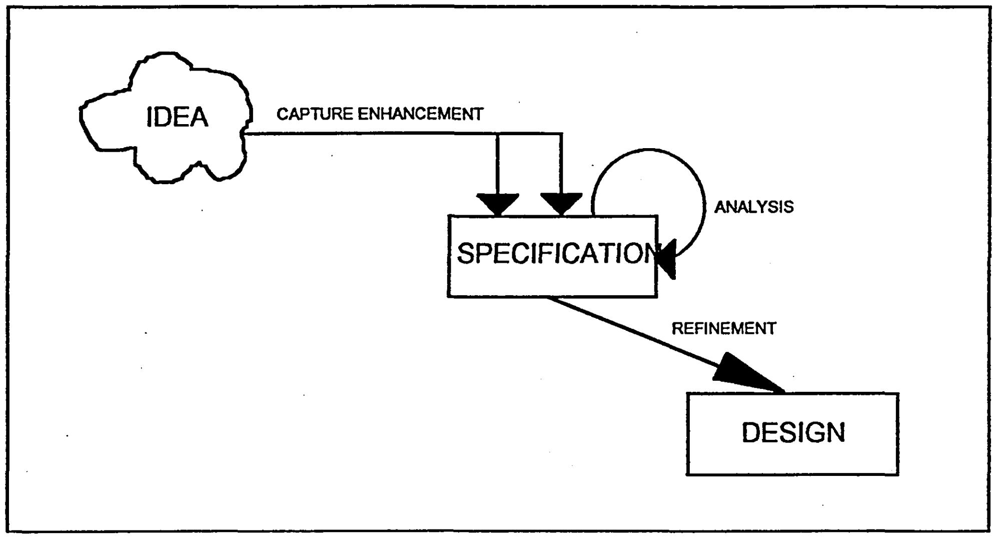


Figure 3.2: Interrelationship of Stages of Software Development using Formal Methods.
Source: Norris, Newman and James, A Step-by-Step Guide to Using Formal Methods, British Telecommunications Engineering, Vol. 5, Jan. 1987.

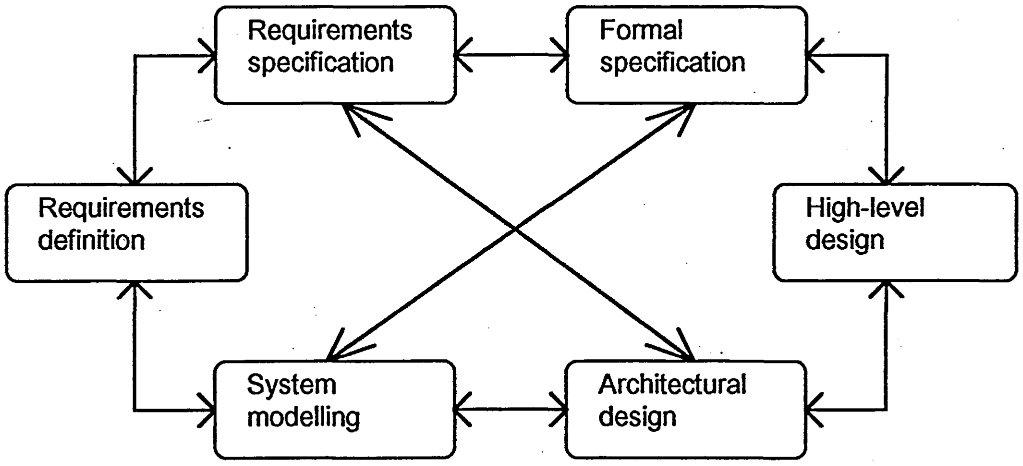


Figure 3.3: Overview of software development incorporating formal specification.
Source: Sommerville, Software Engineering, 4th ed., Addison-Wesley, 1992.

There may be scope for incorporating mathematics into structured methods or integrating formal specification and structured methods. Semmens and Allen [Sem91] and Randell [Ran91] have shown that formal data structures written using the Z notation can be partly derived from data flow diagrams, whilst Stepney [Ste90] has worked on specifying entity relationship diagrams in Z. Polack [Pol91, Pol92] has drawn on this work to propose a draft technique for formalising products produced by SSADM. She sees an opportunity to introduce formal specification to existing Structured Methods users:

"The use of formal notations in the context of structured analysis is seen as valuable in the introduction of formal notations to industrial users. The systems analysis provides diagrams and dictionary definitions about which the Z notation can be structured. This simplifies the production of Z and improves the precision of the system specification."

3.3 The Z Notation

The position of formal methods was certainly strengthened by the development of particular notations designed specifically for use in software engineering. These notations can be divided into three broad categories.

Process algebra, such as CSP [Hoa85] which describes the system as a group of sequential processes. Processes communicate with one another and only certain sequences are permissible.

Algebraic techniques, such as OBJ [Gog88] which model system behaviour divorced from the system state. A particularly abstract approach.

State based techniques including VDM [Jon90] and Z [Spi92, Spi88, Dil90, Wor92]. Both are based on discrete mathematics using typed set theory (sets contain objects of the same type) and Boolean predicate logic. They endeavour to describe a system as a set of possible states with invariants constraining the set of states and preconditions to possible changes between one valid state and another. The Z notation includes a schema calculus, a schema being a collection of constrained name-value pairs within a schema-box (see Fig 3.4). Schemas give a modular feel to a specification and the calculus provides ways of creating connections and procedures via conjunction, disjunction, piping and composition, as well as a means of constructing larger units by including one schema within another. There are conventions, exhibited through the use of certain 'decorations', to give notions of 'before and after' and 'input and output'.

The reader should note that the Z notation refered to here and throughout this thesis is the notation described by Spivey, who's *Z: A Reference Manual* [Spi92] can be refered to for a glossary of Z symbols and names, and a syntax of Z.

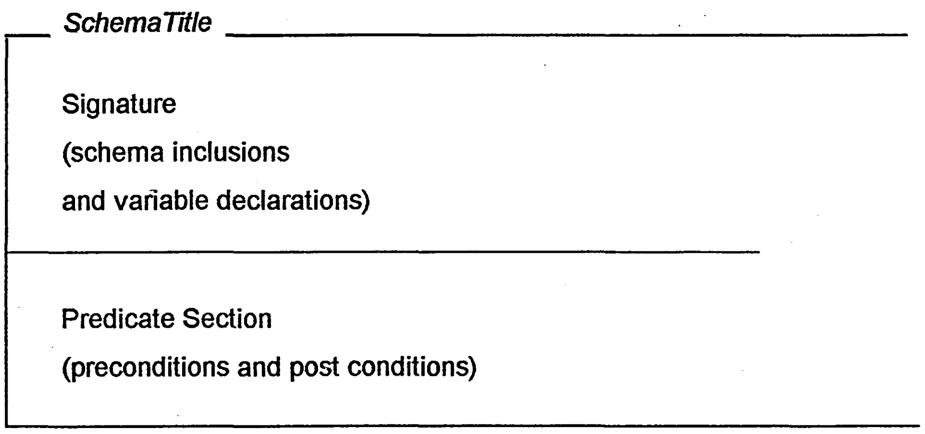

SchemaTitle
Signature
(schema inclusions
and variable declarations)
Predicate Section
(preconditions and post conditions)

Fig. 3.4: A Z Schema.

It may be that with a selection of notations to choose from some are found to be more or less appropriate for different categories of system, or that different notations are used to specify different aspects of a single system.

Formal specification using the Z notation has already been used in diverse applications: for example, Morgan and Suffrin [Mor84] have reported on the specification of a UNIX filestore, Spivey [Spi90] has discussed the specification of a kernel for a real time system, Delisle and Garlan [Del90] have shown how an oscilloscope may be formally specified, AT&T Bell Laboratories [Zav91] have investigated the specification of telephone exchanges, Woodcock *et al.* [Woo94] have specified Defence Standard 00-56 (the British standard for the development of safety critical systems), we have given a generic specification for lift systems [And92] (this and our other published papers referenced in this text are included in a volume of appendices) and, in probably the most well known industrial application of formal specification, IBM have produced, in several volumes, a detailed specification of a Customer Information and Control System.

As a point of interest, IBM have called their formal software engineering *Cleanroom Software Engineering*, by analogy with semiconductor fabrication where defects are avoided by manufacturing in an ultra clean environment, based on the notion that defects in software should be prevented rather than discovered [Mill87].

A survey of Z users [Bar91], carried out as part of the *ZIP project* which is concerned with the enhancement of the use of the Z notation, provides interesting feedback from academic and commercial practitioners, the majority of a positive nature. It contains statistical information, on over 50 projects, concerning size and type of project, length of specifications, iterations undertaken and ratio of mathematical notation to natural language in specifications. The data concerning the use of computer tools was of particular interest to us, and indicated that tool use was common and, in general, the larger the project the larger the use of tools (see Fig. 3.5).

The Alvey Programme, a UK Government venture to stimulate British IT research, also noted and promoted interest in the Z notation and, in its 1987 annual report [Alv87] also mentioned method integration and tool support:

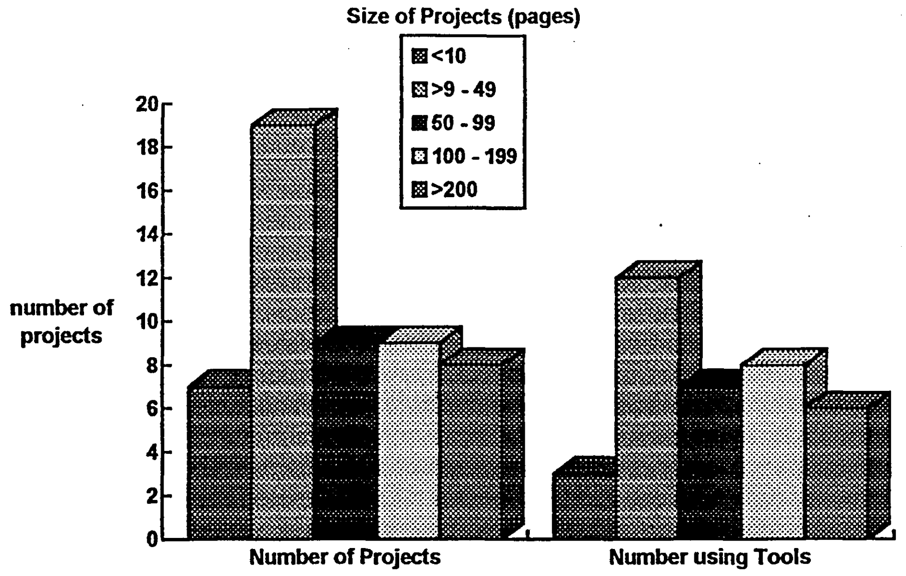


Fig. 3.5: Use of tools compared with size of Z project.
Source: Barden et al., Report of a Survey into the use of Z, Logica, 1991.

"[Formal methods] *continues to be one of the most successful parts of the software engineering programme...The most-used methods appear to be VDM (seen as a mature method at the beginning of the programme); Z, probably now a mature method with the publication of the 'Z handbook'; and LOTOS for protocols. The rapid uptake of these methods by industry and the associated dramatic rise in the number of industrialists trained in these methods is creating feedback which is shaping the direction of future research...the emphasis is towards developing the 'method' aspect of these techniques and, in particular, linking them to existing design methods already widely used in industry and commerce, e.g. SSADM, JSD, etc. However, more advanced applications in the next few years will require fundamental advances in more powerful logics and tools to support them and theorem-proving techniques.*"

It is encouraging to note that *"formal specification, leading to 'animation' and verification"* was cited amongst critical areas of technical development in the Alvey

Programme's original strategy (we discuss animation and describe our techniques for animation in subsequent chapters).

There is no lack of academic and industrial courses offering schooling in the Z notation. In 1991 there were no less than 36 academic institutions offering undergraduate and/or graduate courses, and 18 offering industrial short courses [Nic91].

At Sheffield Hallam we can claim to have as much experience as most in the teaching of Formal Methods; the Z notation has been a part of undergraduate and post-graduate courses at the institution since 1988. Of particular interest, from the point of view of studying the use of Z, has been the author's experience of post-graduate students on the MSc Information Engineering Technology course. These students have had, perhaps, the best opportunity to follow software development from the conception of system requirements through to implementation. Having completed a unit studying the Z notation and its use they have then been able to enhance and put their new skills fully to the test by embarking on a learning contract [And93a].

In essence, a learning contract is a negotiated case study or mini-project. Student individuals or groups are encouraged to suggest systems that they may be interested in developing - part time students are often working in a technical capacity and are usually keen to apply Formal Methods to their areas of expertise. The contract is then divided into a schedule of development stages; the first being the production of a system requirements document, the second the production of an abstract Z specification. Further stages are negotiated and may include one or more stages of refinement, the construction of proofs of consistency within the specification, animation and implementation in a target language such as C.

The experience from the student's and lecturer's point of view has almost always been a positive one and many interesting and practical systems have been developed including an Automated Teller Machine simulation, an electronic components thermal evaluation system, an intelligent multi-storey car parking system, a UNIX style process scheduler, a motorist's route planner, GEORGIS - a rail failure database system for British Rail, a warehouse stock control system and a registration system for an electronic mailbox system for practitioners in the medical professions.

It has been apparent that producing the initial, abstract, specification has invariably been an arduous task, but, once achieved, students have found the later stages of development (in particular, implementation) far more straightforward that expected.

The difficulty found by students in writing Z (especially when only given natural language system requirements) has been particularly apparent on undergraduate courses. Notwithstanding patchy knowledge of software engineering in general, a degree of trepidation when it comes to learning mathematics and a common confusion between programming and specification, students have shown difficulties in thinking in the abstract and then transforming such notions into correct mathematical interpretations. This has been observed from experience in the classroom, marking assignment and examination scripts, and in formal student feedback.

Consideration of such issues in the writing of Z specifications has led us to believe that a simple to use, step by step, method for producing specifications from natural language system requirements, would be of great benefit in enabling students and practitioners to master the mathematical skills required for formal specification.

REFERENCES IN CHAPTER 3

Val87. Valentine, S., *Why Z?*, Systems International (Software Specification), March, 1987.

Wor92. Wordsworth, J. B., *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering*, Addison-Wesley, 1992.

Hoa86. Hoare, T., *Maths adds safety to computer programs*, New Scientist, 18th Sept., 1986.

Inc88. Ince, D., *Z and system specification*, Journal of Information and Software Technology, Vol. 30, No. 3, April, 1988.

Nor91. Norcliffe, A. and Slater, G., *Mathematics of Software Construction*, Ellis Horwood Ltd., 1991.

Sla87. Slater, S. (Ed.), *Essential Mathematics for Software Engineers*, Peter Peregrinus Ltd., 1987.

BBC93. BBC Radio 4, *File on Four* on *Software Engineering*, Oct. 19th, 1993.

For94. Information and Technology Foresight Panel, *The Delphi Questionaire (Phase I)*, UK Technology Foresight Programme, Office of Science and Technology, Aug. 1994

Mai87. Maibaum, T. and Sadler, M., *Formal Methods: A Commentary*, Journal of Information Technolgy, Vol. 2, No. 2, June 1987.

Dji81. Djikstra, E. W., *Forewood* to Gries, D., *The Science of Programming*, Springer-Verlag, 1981.

Nor87. Norris, M. T., Newsman, P. J. and James P., *A Step-by-Step Guide to Using Formal Methods*, British Telecommunications Enginering, Vol. 5, Jan. 1987.

Pla92. Plat, N., van Katwijk, J. and Toetenel H., *Application and Benefits of Formal Methods in Software Development*, Software Engineering Journal, Sept. 1992.

Gib88. Gibbins, P. F., *What are Formal Methods*, Journal of Information and Software Technology, Vol. 30, No. 3, April 1988.

Jac87. Jackson, M., *Power and Limitations of Formal Methods for Software Fabrication*, Journal of Information Technology, Vol. 2, No. 2, June 1987.

Der87. McDermid, J., *The Role of Formal Methods in Software Development*, Journal of Information Technology, Vol. 2, No. 3, Sept. 1987.

Jon90. Jones, C. B., *Systematic Software Development using VDM*, 2nd Edition, Prentice Hall International, 1990.

Ear86. Earl, A. N., Whittington, R.P., Hitchcock, P. and Hall, A., *Specifying a semantic model for use in an integrated project support environment*, in *Software Engineering Environments* (Sommerville, I., Ed.), Peter Peregrinus Ltd., 1986.

Hay87. Hayes, I., (ed.), *Specification Case Studies*, Prentice Hall International,1987.

Kin89. King, S., *Z and the Refinement Calculus*, Procs. of VDM90 Symposium, Sept. 1989.

Lit92. Litteck, H. J. and Wallis, P. J. L., *Refinement methods and refinement calculi*, Software Engineering Journal, Vol. 7, No. 3, May 1992.

Som92. Sommerfield, I., *Software Engineering*, 4th Edition, Addison-Wesley, 1992.

Spi92. Spivey, J. M., *The Z Notation: A Reference Manual*, 2nd Edition, Prentice Hall International, 1992.

Hoa85. Hoare, C. A. R., *Communicating Sequential Processes*, Prentice Hall, 1985.

Gog88. Goguen, J. A. and Winkler, T., *Introducing OBJ 3*, SRI International, 1988.

Spi88. Spivey, J. M., *Understanding Z: A Specification language and its formal semantics*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1990.

Dil90. Diller, A., *Z: An Introduction to Formal Methods*, John Wiley and Sons,1990.

Sem91. Semmens, L. and Allen, P., *Using Yourdon and Z: an approach to formal specification*, Proceedings of the fith annual Z User Meeting, Dec. 1990.

Ran91. Randell, G., *Data Flow Diagrams and Z*, Proceedings of the fith annual Z User Meeting, Dec. 1990.

Ste90. Stepney, S., *Specifying entity relationship diagrams in Z*, ORCA/Logical, 1990.

Pol91. Polack, F., Whiston, M. and Hitchcock, P., *Structured Analysis - A Draft Method for writing Z Specifications*, Proceedings of the sixth annual Z User Meeting, Dec. 1991.

Pol92. Polack, F., *Integrating formal notations and systems analysis: using entity relationship diagrams*, Software Engineering Journal, Sept. 1992.

Mor84. Morgan, C. and Sufrin, B., *Specification of the UNIX filing system*, IEEE Trans. Software Engineering, Vol. 10, No. 2, 1984.

Spi90. Spivey, J. M., *Specifying a real-time kernel*, IEEE Software, Vol. 7, No. 5, 1990.

Del90. Delisle, N. and Garlan, D., *A formal specification of an oscilloscope*, IEEE Software, Vol. 7, No. 5, 1990.

Zav91. Zave, P. and Jackson, M., *Techniques for Partial Specification and Specification of Switching Systems*, Proceedings of the sixth annual Z User Meeting, Dec. 1991.

Woo94. Woodcock, J. C. P., Gardiner, P. H. B. and Hulance, J. R., *The Formal Specification in Z of Defence Standard 00-56*, Formal Systems (Europe) Ltd, 1994.

And92. Andrews, S. J., *A Lift System*, a case study in *A Z Readers Course*, Shefield Hallam University Pavic Publications in Assn. with the D.T.I., 1992.

Mil87. Mills, H. D., Dyer, M. and Linger, R., *Cleanroom Software Engineering*, IEEE Software, Vol. 4, No. 5, 1987.

Bar91. Barden, R., Stepney, S. and Cooper, D., *Report of a Survey into the use of Z*, ZIP document , Logica Ltd., Dec. 1991.

Alv87. *Alvey Programme Annual Report*, Alvey Directorate, IEE Publishing, 1987.

Nic91. Nicholls, J.E., *A Survey of Z Courses in the UK*, in Proceedings of the *1990 Z User Workshop*, Springer-Verlag, 1991.

And93a. Andrews, S., J., *Learning Contracts*, a case study in *Innovations in Mathematics Teaching*, Staff and Educational Development Association Publications, 1993.

And93b. Andrews, S., Edwards, P., Faulkner, B., Hodgkin, L., Norcliffe, A., Smith, F., and Steere, P., *Student Feedback as an Element in Assuring Course Quality*, Proceedings of the 1993 Undergraduate Mathematics Teaching Conference, Shell Centre for Mathematical Education, 1993.

CHAPTER 4: WRITING Z - THE OPERATOR METHOD

Life is too short to learn German.
Richard Parson, British classicist (1759 - 1808).

4.1 Introduction.

Although most university computing courses now include a study of the Z notation, the teaching of Z, as we noted at the end of the previous chapter, is still not without its problems. Students not only find difficulties coming to grips with the notation and the underpinning mathematics, but experience enormous problems when they first come to use the notation to construct a system specification from given requirements. Students find it extremely hard coping with abstraction and identifying the particular variables that make up the state schema. Once through this abstraction bottleneck, and having produced the state schema, students find it much easier to build the associated operation and error schemas, and go on to complete the specification. They may not immediately specify the operations correctly, but they at least seem more comfortable with this part of the process - probably because it is more mechanical and there is less need of abstraction.

If students in academia experience these problems it is more than likely that software engineers in industry, when being trained in the use of formal methods, will experience similar problems. Not only does this impede the technology transfer process, but it makes it difficult to identify exactly what the technology is that is being transferred, other than abstraction and the ability to create mathematical models. Such abilities are learnt slowly and come only gradually with experience. The problems of replacing experienced staff when they move on, when a transferable technology or systematic method is unavailable, mitigates against the adoption of formal methods by industry.

The lack of a systematic method for developing Z specifications also means that tool support for the process is problematic. The type and syntax checkers currently available do not really assist the trainee software engineer to construct Z specifications, although they are of tremendous help to the experienced Z user.

In the previous chapter we mentioned the idea of integrating formal and structured methods and indicated some of the exploratory work that has been done in this area. This may well become an accepted approach and is certainly palatable to the software engineering industry. Once well defined, such methods will no doubt enter the mathematics curricula of computing courses and would be suitable for students already familiar with the systems analysis in methods such as SSADM and Yourdon. Nevertheless formal specification, and the mathematical notations thereof, form valuable disciplines for study in their own right, and we feel that a simple approach to enable students to progress from natural language requirements to a mathematical representation is what is needed.

The issue with the methods integration work of Semmens and Allen [Sem91], Randell [Ran91] and Polack et al. [Pol93] is that something verging on a full blown structured approach has to be carried out first. Admittedly there is tool support for this, such as SELECT and ASCENT, but a detailed knowledge of the structured approach being adopted is required, and carrying out a full scale structured approach, whilst beneficial, can be time consuming. Further, if the end product of these endeavours is a formal specification written in Z, then it has to be accepted that the Z produced by converting, for example, Yourdon diagrams, will not be as abstract or as simple as it might otherwise have been. Examples in Semmens and Allen [Sem91] bear this out, although the Z has been produced in a systematic fashion. Arguably, the ability to reason in abstract terms with the resulting Z has then been reduced by the complexity of the Z. Also the ability to animate the resulting Z is made harder.

What is needed is a simple approach which enables the specifier to progress from the user requirements to the Z in a systematic but direct way. The OPERATOR method described in this chapter [And95] was developed with this goal in mind. Although as a method it draws on the well-proven ideas of structured methods, it does not require a full scale structured analysis to be carried out first and is essentially free standing.

4.2 The OPERATOR method - a simple example.

To see how the OPERATOR method works we use it to produce the state schema needed in the specification of a simple security system. Assume the system we are to specify has the following user requirements.

> *"The system is to monitor the whereabouts of staff in an organisation. The organisation is located in its own building and, as staff check themselves in and out of the building, the system notes whether they are in or out as appropriate. The system can be queried at any time to see who is in or out, and must cope with staff joining and leaving the organisation."*

The word OPERATOR is an acronym with the letters standing for Objects, Properties, Entities, Relationships, Assemble, Trim, Other and Repeat. Step 1 of the method therefore begins by identifying the **objects** that make up the system. In our example obvious candidates for objects are the staff who work in the building. Whilst it is not imperative that all objects be identified at this stage - indeed, the later identification of objects is an integral part of the method - it is worth noting that there are no other obvious objects making up the system that need concern us. It is worth noting, too, that we need not be overly strict about what constitutes an object other than that objects should be nouns and have some concrete existence [Sul93].

Step 2 of the method requires us to identify the **properties** of these objects. At this stage it is important to note that we are looking only for simple has/have properties. Other relationships are established during step 4 of the method. From the requirements of our system it is clear that staff have whereabouts, and it is this property that the system must monitor. Staff in the organisation have no other properties of significance and thus we can proceed to step 3 and identify the entities making up the system.

The **entities** of the OPERATOR method are the nouns identified in steps 1 and 2. The entities are thus the staff and their whereabouts. As part of this third step we must also describe the entities in terms of the Z notation. Basic types are therefore needed and we parachute in the type set *STAFF_ID* and introduce the enumerated type *IN_OUT* containing the elements in and out. The system entities *staff* and *whereabouts* are thus declared as follows:

$$\begin{array}{lll} staff & : & \mathbb{P} STAFF\_ID \\ whereabouts & : & \mathbb{P} IN\_OUT \end{array}$$

As system entities *staff* and *whereabouts* are sets of *STAFF_ID* and *IN_OUT* values respectively, thus explaining the use of the powerset symbol in the declarations. The variables *staff* and *whereabouts* are possible state variables; additional state variables are identified via step 4 of the method, where relationships between system entities are established.

Relationships between entities are identified in a systematic way using the concept of the entity/entity matrix shown below. At this stage the aim is to identify binary relations only. More complicated relationships are introduced via the data invariant of the state schema once all state variables have been identified.

	staff	*whereabouts*
staff	-	*location_of*
whereabouts	*located*	-

In each cell of the matrix are put names of relevant binary relations between the pairs of entities involved. The assumption is that the entity in the row of the matrix is the domain of the relation, and the entity in the column is the range. Where system entities are not sets but single elements, they should be regarded as singleton sets if binary relations involving them are needed. In practice this seldom happens. Thus, *location_of* is a binary relation between staff and their whereabouts and, since at any time staff have unique locations, then the binary relation is actually a partial function with the following declaration:

$$location\_of \quad : \quad STAFF\_ID \rightarrowtail\mkern-14mu\shortmid\; IN\_OUT$$

The binary relation *located* is not a function as several staff may be in or out of the building at any one time. Its declaration is this:

$$located \quad : \quad IN\_OUT \leftrightarrow STAFF\_ID$$

We should note that the remaining cells of the matrix are empty because no relevant relations exist between the entities concerned.

Step 5 of the method is to **assemble** the list of candidate state variables. This list contains the system entities together with the binary relations identified. The assembled list of variables and their declarations is thus as follows:

$$\begin{array}{lll} staff & : & \mathbb{P}\ STAFF\_ID \\ whereabouts & : & \mathbb{P}\ IN\_OUT \\ location\_of & : & STAFF\_ID \rightarrowtail\mkern-14mu\shortmid\; IN\_OUT \\ located & : & IN\_OUT \leftrightarrow STAFF\_ID \end{array}$$

It is more than likely that this list is longer than it need be and, so, in step 6 of the method we **trim** it down. The trimming is achieved systematically by getting rid of redundant information. We can usefully note that *staff* is the same as dom *location_of*. Thus, if we wish we need not include *staff* in the state schema provided we include the partial function *location_of*. Similarly, we need not include the set *whereabouts* because this is the same as ran *location_of*. Finally, we need not include *located* because this is just the inverse of *location_of*.

In theory, then, all the information we might need is contained within the *location_of* function. However, it may be sensible to include *staff* in the abstract specification even though the information is redundant, so that a direct record of the users of the building is ready to hand for specification purposes. The level of redundant information is really a matter of taste. Clearly it should not be great, but at the same time it is important to ensure that specifications are readable and easily understood [Gra91,Spi92]. The trimmed list of state variables is thus:

$$
\begin{array}{lll}
\mathit{staff} & : & \mathbb{P}\ \mathit{STAFF\_ID} \\
\mathit{location\_of} & : & \mathit{STAFF\_ID} \mathrel{+\!\!\!\rightarrow} \mathit{IN\_OUT}
\end{array}
$$

The remaining 2 steps of the method require us to check whether there are **other** objects of note, and to **repeat** the process with them included. Fortunately there are no other objects and therefore no need to repeat the process. Repeating the process is in principle not difficult. Care should be taken to check for additional relationships between new and existing entities during the repeat step 4.

The culmination of the OPERATOR method is thus the listing of state variables given above. The state variables and the properties which they possess can now be set down in the system state schema:

```
 ______ System ________________________________________
|
| staff : ℙ STAFF_ID
| location_of : STAFF_ID ⇸ IN_OUT
|______________
|
| staff = dom location_of
|______________________________________________________
```

The OPERATOR method will not determine the data invariant. However, having identified the state variables, the data invariant can be determined from knowledge about the system and the Z constructs used to model the system variables.

From here on, the rest of the system specification can be established. This will include specifying state changing operations such as checking in and checking out of the building by staff, adding new staff members and removing staff from the system when, for example, they leave the organisation. Querying operations, which do not change the state, can similarly be specified, and might include such operations as querying the system to see who is in or out of the building.

Such state changing and querying operations will not be specified here to save time. There is nothing complicated about their specification and given the system state schema they are easily produced. We shall, however, revisit the specification of system operations after the OPERATOR method has been enhanced by the inclusion of a diagramming notation and the means of addressing system complexity.

4.3 Another simple example.

To demonstrate the applicability of the OPERATOR method we consider, albeit briefly this time, another example - that of a simple banking system. The requirements of the system, we assume, are the following.

> *"The balances and overdraft limits of accounts at a bank are to be monitored by the system. Account holders can make deposits and withdrawals and, if they have sufficient funds, can change their overdraft limits. As well as furnishing information on balances and overdraft limits, the system should cope with opening and closing accounts."*

Application of the OPERATOR method, with brief annotations, is as follows.

Objects:	*accounts*, *holders*
Properties:	*accounts* have *balances*
	accounts have *od_limits*
Entities:	*holders* : $\mathbb{P}$ *HOLDER_ID*
	accounts : $\mathbb{P}$ *ACC_NO*
	balances : $\mathbb{P}\mathbb{Z}$
	od_limits : $\mathbb{P}\mathbb{Z}$

Here we should note that the basic type, $\mathbb{Z}$, representing the integers, is being used to model the balances and overdraft limits (in pence) of individual accounts. Other types used have their obvious meanings.

Relationships:

	holders	accounts	balances	od_limits
holders	-	account_of	-	-
accounts	holder_of	-	balance_of	od_limit_of
balances	-	-	-	-
od_limits	-	-	-	-

Assemble:

holders	:	ℙHOLDER_ID
accounts	:	ℙACC_NO
balances	:	ℙℤ
od_limits	:	ℙℤ
account_of	:	HOLDER_ID ⇸ ACC_NO
holder_of	:	ACC_NO ↔ HOLDER_ID
balance_of	:	ACC_NO ⇸ ℤ
od_limit_of	:	ACC_NO ⇸ ℤ

Note that the concept of joint accounts is being modelled by declaring *holder_of* to be a binary relation and not a partial function. By declaring *account_of* to be a partial function, the assumption is that holders can only hold one account.

Trim:

holders	:	ℙHOLDER_ID
account_of	:	HOLDER_ID ⇸ ACC_NO
balance_of	:	ACC_NO ⇸ ℤ
od_limit_of	:	ACC_NO ⇸ ℤ

In trimming the list we have noted that *accounts* is dom *balance_of*, that *balances* is ran *balance_of*, that *od_limits* is ran *od_limit_of*, and that *holder_of* is the inverse of the function *account_of*.

Other: There are no other objects of note.

Repeat: This step is unnecessary.

The state schema, with its appropriate data invariant, is as follows:

Bank

$holders : \mathbb{P}\,HOLDER\_ID$

$account\_of : HOLDER\_ID \nrightarrow ACC\_NO$

$balance\_of : ACC\_NO \nrightarrow \mathbb{Z}$

$od\_limit\_of : ACC\_NO \nrightarrow \mathbb{Z}$

$holders = \mathrm{dom}\ account\_of$

$\mathrm{ran}\ account\_of = \mathrm{dom}\ balance\_of$

$\mathrm{dom}\ balance\_of = \mathrm{dom}\ od\_limit\_of$

$\forall x : \mathrm{dom}\ balance\_of \bullet$

$(balance\_of(x) \geq od\_limit\_of(x)$

$\wedge\ od\_limit\_of(x) \leq 0)$

Note that the data invariant is reflecting the operating assumptions of a normal bank - namely that all accounts have balances and overdraft limits, that overdraft limits should not be exceeded, and that overdrafts represent negative amount of cash. Once again, to complete the specification, operations that change the state of the system, and those which only query the state, would now be specified.

4.4 Using the Method with Students.

The OPERATOR method as it has been described was the prototype of the method which now exists. The prototype was enhanced and extended following testing of the method with students. Here we now describe our experience of using the OPERATOR method in the classroom.

The method was first tested on second year Computing Mathematics degree students at Sheffield Hallam University. Students had already been exposed to discrete mathematics and the Z notation, and were familiar with reading Z specifications. They had, for example, studied the video-based *A Z Readers Course* produced at Sheffield [Coo92] and knew how specifications were structured. They had not, however, had any experience of writing specifications and the OPERATOR method was the first systematic approach they had used to develop Z specifications.

Working in small groups students had to specify a simple library system. An extract from the given user requirements document is as follows:

> *"In order to monitor who the users of the library are, which copies of books they have on loan, and which copies are available for borrowing, a simple computer-based system is to be developed. Any copy of a book that has been borrowed will have a return date stamped inside it and this will be noted by the system. The system must also log the acquisition of new copies of books and note their removal, and should enable new users to join the library and existing users to leave."*

The marks for the complete (non-robust) specification were 50, of which 10 were available for use of the OPERATOR method to determine the list of state variables and their declarations. The average mark for use of the OPERATOR method was 7.41 with a standard deviation of 1.57. The marks ranged from 4 to 9 and there were 17 groups of students. Most succeeded in using the method well and produced a variety of consistent specifications. Most lists of state variables were variations on the following:

users	:	$\mathbb{P}$*USER_ID*
copies	:	$\mathbb{P}$*COPY_ID*
books	:	$\mathbb{P}$*BOOK*
borrower_of	:	*COPY_ID* $\nrightarrow$ *USER_ID*
book_of	:	*COPY_ID* $\nrightarrow$ *BOOK*
status_of	:	*COPY_ID* $\nrightarrow$ *STATUS*
duedate_of	:	*COPY_ID* $\nrightarrow$ *DATE*

Several groups had been harsher with their trimming than others and had removed *copies* and *books*. Others had introduced the concept of library cards and additional information about books such as their titles and authors. A common omission was the *status_of* function which indicates whether a book is available for borrowing or not. Since its inclusion in the specification is not essential, the omission was not penalised.

In summary, students found the method easy to understand and simple to use. The method had been demonstrated using the examples considered in Sections 2 and 3, and students were able to apply the ideas readily to develop the simple library system. Many of the specifications turned out similar as a result of applying the method, although there had been minimal copying of ideas by groups. Whether this high level of reproducibility is a good feature of the method is debatable. The approach certainly steers the specifier towards the use of functions and relations when perhaps simpler structures might have been used. The security system, for example, is easily developed in terms of just sets [Nor91, Coo92]. Students commented that they found the method enabled them to construct specifications in a systematic way. In general they found this helpful and were able to have sensible discussions about the system based around the approach being adopted.

Although the comments of the students were positive in the main, the method does have its limitations. The approach, though systematic, is still very abstract. It is interesting to note that some students were drawing informal diagrams to help them apply the method. Given that the success of structured methods such as SSADM, Jackson, and Yourdon seem to hinge on the use of accompanying diagrams, the author and his director of studies deemed it necessary that the OPERATOR method should also have a diagramming notation. Not only would this help the specifier with the process of abstraction, but the diagrams would be of potential help in communication with the would be client or user about the essential features of the system to be built. In the next section we therefore show how the method was enhanced.

As well as being abstract, the approach as outlined so far does not really address system complexity. In discussions, the students commented that they felt the method could soon become unworkable if the number of entities became large. Drawing up a large entity/entity matrix would be difficult, for example, and ensuring that the data invariant of a large state schema was correct would also not be easy. In Section 6 we therefore show how the OPERATOR method, and its diagramming notation, can be extended to address system complexity and to embrace structural considerations such as partitioning a system into several subsystems.

4.5 Enhancing the method with a graphical front end.

The graphical notation described in this section has been developed to accompany the method and to facilitate the O, P and E stages. Its use is therefore designed to help identify the objects and entities that make up the system being specified. The notation is as follows:

- The system at the top level is represented by an appropriate descriptor written inside a rectangular box as shown:

The convention is that the first letter of the descriptor is an upper case letter. If we were developing a banking system we may well expect to see *Bank* written inside the box instead of *System*.

- Objects and other system entities, related by the has/have property, are also represented by names written inside rectangular boxes:

$$\boxed{\mathit{staff}}$$

The convention with system entities (objects are also entities) is that their names are written in lower case throughout.

- Each of the boxes representing an entity has the set, to which the entity belongs, written alongside in Z, e.g.:

$$\mathbb{P}\ \mathit{STAFF\_ID}$$

The convention here is that types and other sets used are written in upper case letters throughout, and are not contained inside boxes.

- The hierarchical relationships between the above are represented by arrows of appropriate kinds:

——▶ links the system at the top level to the objects out of which the system is comprised.

$\longrightarrow$ links objects to entities, and entities to their associated entities as appropriate. The arrow characterises the has/have property.

$--\!\!\!\rightarrow$ links entities (including those identified as objects) to the types and sets in Z to which they belong.

To show how the notation works, let us draw the diagrams that represent the security, banking and library systems considered earlier. Fig. 4.1 shows the simple security system.

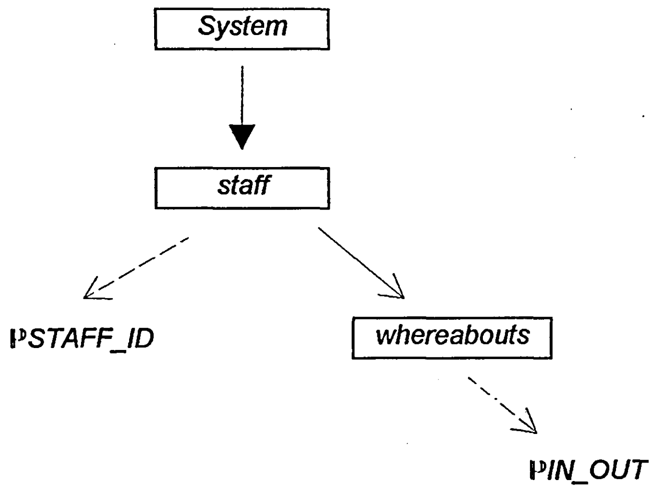


Fig. 4.1 - Graphical representation of the simple security system

The diagram tells us that the system state at the highest level is called *System*. The objects in the system are *staff* who have *whereabouts*. The system entities are therefore *staff* and *whereabouts*, and these are possible state variables. The variable *staff* is a member of the constructed type set ℙ*STAFF_ID*, and *whereabouts* belongs to the constructed type set ℙ*IN_OUT*.

Fig.4.2 shows the banking system and Fig. 4.3 represents one interpretation of the simple library system. If we take Fig. 4.3, for example, this is telling us that the library is comprised of *users* and *copies*, which are being regarded as the objects of the system. The sets *users* and *copies* are sets of *USER_ID* and *COPY_ID* values respectively. Properties of *copies* are that they have associated *books*, *locations* and *dates* stamped in them. The sets *books*, *locations* and *dates* are sets of *BOOK*, *STATUS* and *DATE* values respectively.

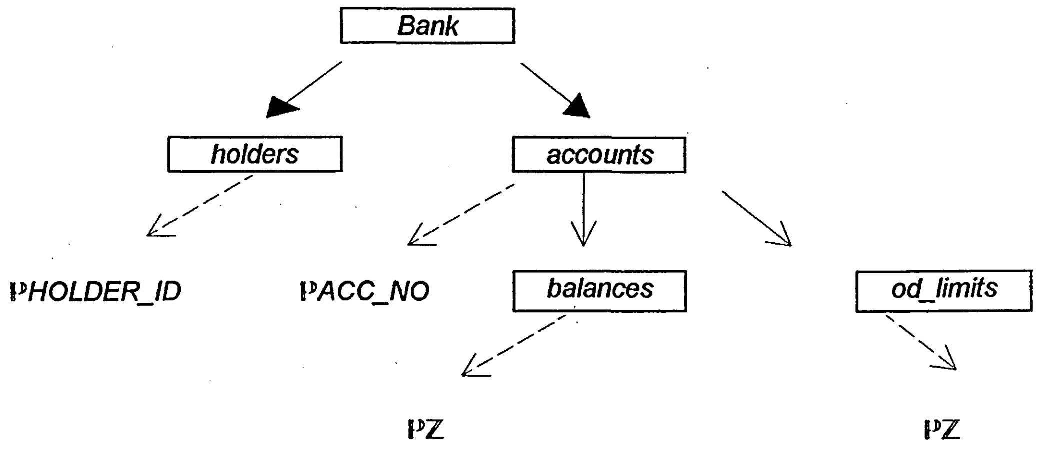


Fig. 4.2: Graphical representation of the banking system

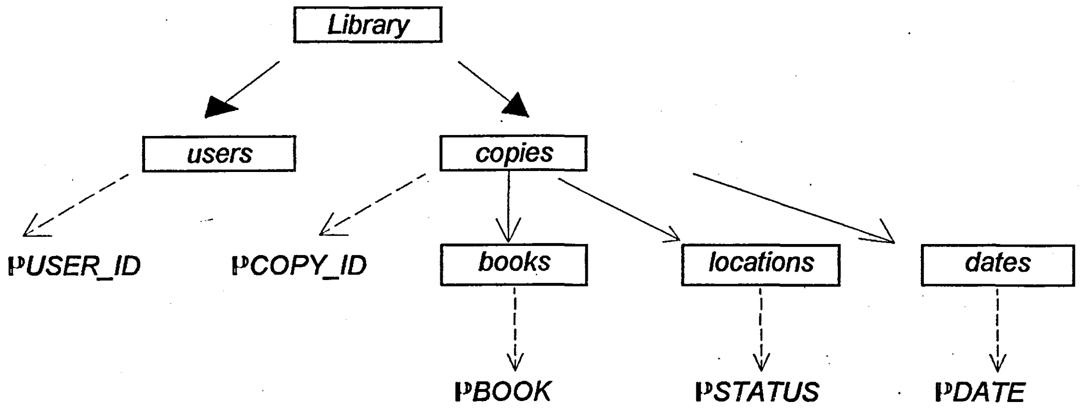


Fig. 4.3: Graphical representation of the library system

Hopefully, the diagrams speak for themselves. It should be noted that different diagrams may well lead to the same Z specification. In Fig. 4.2, for example, it is assumed that *holders* and *accounts* are both objects. There is nothing wrong with a diagram that regards just *holders* as objects and *accounts* as associated entities - in the sense that account holders have bank accounts. Since *holders*, *accounts*, *balances* and *od_limits* emerge as the system entities either way round, the odds are that the resulting

Z specifications of the state will be the same. The prime purpose of the diagramming notation is to assist the specifier to identify system entities, and this we feel it does. The strength of the notation is that it is graphical and hierarchical, and readily enables a picture of the system state to be created showing explicitly the entities that are part of it.

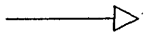
4.6 Addressing System Complexity.

Unless a method can be used to develop a large system, and therefore cope with complexity, it is really no method at all. In this section we show how the method and the associated graphical notation has been extended to cope with the specification of complex systems. The ideas in this section are relatively new and have not been tested out with students for their ease of use. The graphical notation presented in the previous section has, however, been taught to students. For the simple systems considered, the notation proved to be quite adequate and was apparently easy to teach and easily learnt.

The extended notation described here has been used to develop structured systems (for example a vending machine and a realistic library system coping with loans and reservations) with considerable success and we are confident that in its extended form the OPERATOR method will stand up well in future trials with students.

Complexity is addressed by partitioning a system into appropriate subsystems and applying the OPERATOR method to each subsystem in a 'divide and conquer' fashion. To do this the diagramming notation requires a new kind of rectangular box and a new kind of arrow. The new kind of box is one containing a subsystem name. Thus, in the case of the simple library system we considered in section 4, if it were felt that a partitioning of the system into three subsystems, namely *Users*, *Copies* and *Loans*, was needed, then a typical subsystem box would be the following:

The new arrow that is needed is the following one:



which links a system to its subsystems.

To see how the ideas can be applied, let us revisit the library and think of it not as a monolithic system, with no real structure, but comprising the three subsystems proposed above. This view of the library is illustrated diagrammatically in Fig. 4.4, where the extended subsystem notation is used.

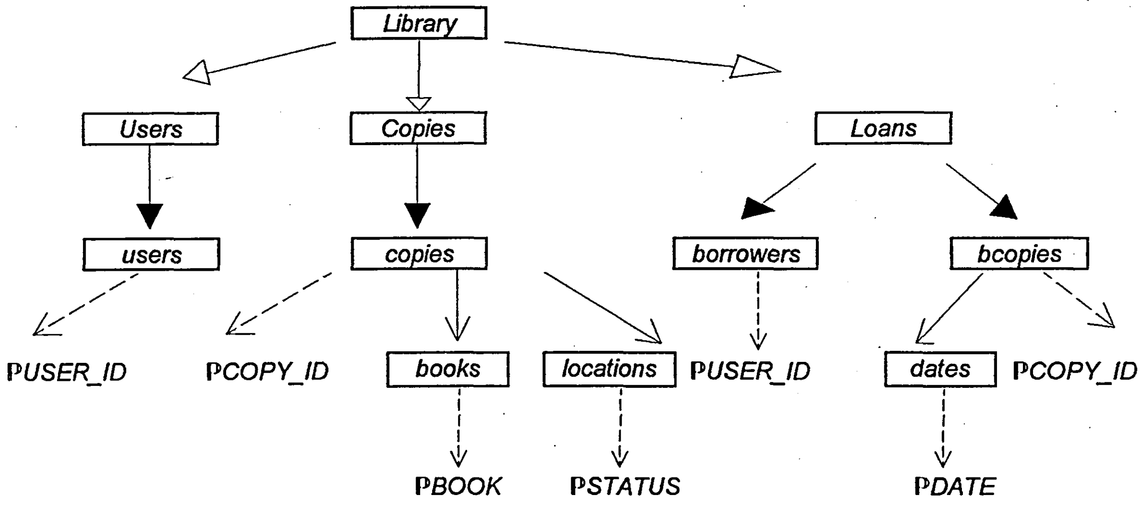


Fig. 4.4: Partitioned view of the library system

The OPERATOR method can now be used to develop substate schemas to specify the states of the *Users*, *Copies* and *Loans* subsystems. The state schema, *Library*, is then the schema which includes these three substate schemas. Application of the OPERATOR method, as described earlier, leads to the following *Users*, *Copies* and *Loans* substate schemas. Their derivation is straightforward and they are presented without explanation. In the *Loans* subsystem note that new variables *borrowers* and *bcopies* (borrowed copies) have been introduced.

Users

$users : \mathbb{P}\, USER\_ID$

Copies

$copies : \mathbb{P}\, COPY\_ID$
$books : \mathbb{P}\, BOOK$
$book\_of : COPY\_ID \nrightarrow BOOK$
$status\_of : COPY\_ID \nrightarrow STATUS$

$copies = \mathrm{dom}\ book\_of$
$\mathrm{dom}\ book\_of = \mathrm{dom}\ status\_of$
$books = \mathrm{ran}\ book\_of$

Loans

$borrowers : \mathbb{P}\, USER\_ID$
$bcopies : \mathbb{P}\, COPY\_ID$
$borrower\_of : COPY\_ID \nrightarrow USER\_ID$
$duedate\_of : COPY\_ID \nrightarrow DATE$

$borrowers = \mathrm{ran}\ borrower\_of$
$bcopies = \mathrm{dom}\ borrower\_of$
$\mathrm{dom}\ borrower\_of = \mathrm{dom}\ duedate\_of$

These three schemas can now be included into one *Library* schema to create the state schema for the library system. The data invariant serves to relate all the variables involved defining, in particular, the status of copies of books that have been borrowed, and those which should be available for borrowing:

Library

Users
Copies
Loans

$borrowers \subseteq users$
$bcopies \subseteq copies$
$\forall c : bcopies \bullet status\_of(c) = borrowed$
$\forall c : copies \bullet c \notin bcopies \Rightarrow status\_of(c) = available$

By contrast, and for comparison, the state schema of the monolithic unpartitioned library system, again developed using the OPERATOR method, is as follows:

Library

$users : \mathbb{P} USER\_ID$

$copies : \mathbb{P} COPY\_ID$

$books : \mathbb{P} BOOK$

$borrower\_of : COPY\_ID \nrightarrow USER\_ID$

$book\_of : COPY\_ID \nrightarrow BOOK$

$status\_of : COPY\_ID \nrightarrow STATUS$

$duedate\_of : COPY\_ID \nrightarrow DATE$

$\mathrm{ran}\ borrower\_of \subseteq users$

$\mathrm{dom}\ borrower\_of \subseteq copies$

$\mathrm{dom}\ book\_of = copies$

$\mathrm{ran}\ book\_of = books$

$\mathrm{dom}\ book\_of = \mathrm{dom}\ status\_of$

$\mathrm{dom}\ duedate\_of = \mathrm{dom}\ borrower\_of$

$\forall c : \mathrm{dom}\ duedate\_of \bullet status\_of(c) = borrowed$

$\forall c : \mathrm{dom}\ book\_of \bullet c \notin \mathrm{dom}\ duedate\_of \Rightarrow status\_of(c) = available$

The 'divide and conquer' approach can now be seen to be working. Partitioning of the system has meant that the diagrams for *Users*, *Copies* and *Loans* are each fairly simple. Indeed these diagrams could have been drawn separately instead of on one diagram as in Fig. 4.4. The overall partitioned view of the library could well have been simply the *Library* box together with the *Users*, *Copies* and *Loans* boxes. Accompanying this would then have been the three subsystem diagrams.

Similarly, partitioning has meant that application of the OPERATOR method to each subsystem now becomes simpler than its application to the monolithic unstructured system. The resulting substate schemas bear this out.

Clearly, were the library system more complicated than the simple one considered here, application of the original OPERATOR method would begin to become unworkable as entity/entity matrices grew in size and the number of binary relations expanded also (as the square of the number of entities). Keeping check on such large monolithic systems would be difficult when it came to trimming and then establishing the total system data invariant.

4.7 System Operations.

So far the whole emphasis of OPERATOR has been on the systematic construction of the system state schema. As we pointed out in the beginning it is usually this initial part of a Z specification which is hardest to write. Having obtained the system state schema, operations that change or query the state can usually be specified without too much difficulty.

The specification of system operations is, however, a very important (and time consuming) part of any specification and where diagrams can be used to help then they should be employed.

A very simple diagramming notation, akin to data flow diagrams in structured methods, can in fact be used to complete the OPERATOR diagramming notation. With reference to the library system of section 4, an operation diagram representing the *BorrowCopy* operation can be drawn as shown in Fig. 4.5.

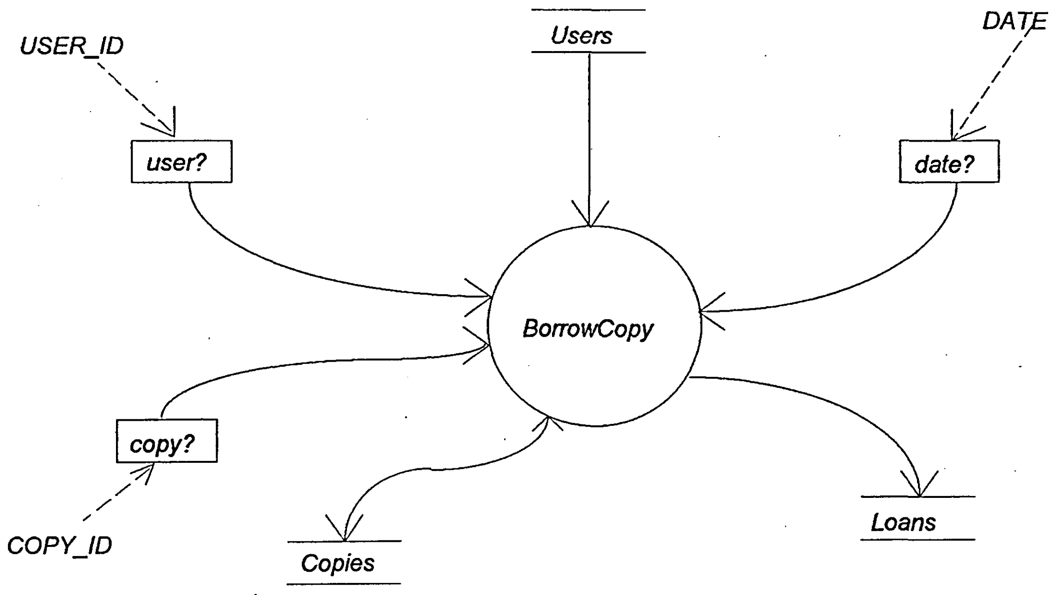


Fig. 4.5: Diagram showing the *BorrowCopy* operation in the library system.

The notation used is deliberately similar to that used to represent data flow diagrams. The operation name is put inside a circle as shown, as in Yourdon, for example. Inputs and outputs are drawn inside rectangular boxes. These are similar to the terminators of data flow diagrams in Yourdon. The sets from which inputs and outputs are drawn are indicated as shown. The dotted arrow is again used to indicate that data modelling is being used. The direction of the dotted arrow will signify whether an input or an output is being modelled. An output arrow would go into the output box and the output itself would have a shriek mark decoration - as in Z.

The system substates that are affected or needed by the operation are represented in the same way that data stores are in Yourdon. The solid arrows are also significant. Thus we see that the *Users* substate will not be changed by the operation. Its contents are only read. The *Copies* substate is both read and written to as the status of the borrowed copy will be changed to borrowed. The *Loans* substate is not read, but is written to.

In the same way that the system entity diagram can be transformed into Z using the R, A, T, O, R part of the method, so too can the operation diagram be turned into Z, once the state schema has been produced. The diagram will enable the specifier to write down immediately what the signature of the operation schema is. The signature of the BorrowCopy operation is thus the following:

```
┌──────── BorrowCopy ─────────────────────────────────
│
│  ΔLibrary
│  ΞUsers
│  ΔCopies
│  ΔLoans
│  users? : USER_ID
│  copy? : COPY_ID
│  date? : DATE
│
├──────────────
│
```

ΔLibrary is needed because we wish to bring into scope all the before and after states of the library and their collective properties. *ΞUsers* is included to alert us to the fact that the state variables in the *Users* subsystem are not being changed. *ΔCopies* and *ΔLoans* are, strictly speaking, not needed because the appropriate before and after states of the *Copies* and *Loans* subsystems are already in scope. Their inclusion alerts us explicitly to the fact that these subsystems are changed by the operation. The inputs, *users?*, *copy?* and *date?* are those indicated on the operation diagram.

As with the production of the state schema previously, the use of the diagram does not help with the predicate part of the operation schema. This, however, can be written and systematically produced with reference to preconditions and postconditions.

4.8 Discussion.

In this chapter we have traced the development of the OPERATOR method from its initial abstract but systematic approach to developing the system state schema to the fuller form it now takes with its diagramming front end and its system partitioning mechanism. Of note is the fact that the approach does comprise a method for going systematicaly from the system entities, identifiable graphically, to the system state schema in a way that addresses complexity in large systems. Of note also is the way in which the OPERATOR approach addresses the strengths and weaknesses of mathematics as a specification language.

Initially the diagramming notation allows the specifier to work interactively with the client to capture the essential features of the system state, including any key structural issues. Boxes and arrows are intuitively simple to work with and model well the hierarchical properties of the system. Data modelling (via the dotted arrows) does not have to involve the client and nothing is lost by not including the data modelling at this stage. Operations can also be represented as operation diagrams and drawn up with the client with direct reference to only the system entity diagrams. Again, data modelling via the dotted arrows does not have to feature on the diagrams at this stage. The diagrams thus serve to help the specifier through the abstraction bottleneck [Nor93] and at the same time facilitate effective communication at a crucial time with the system user or client.

Once the specifier and client are happy that the system requirements are being captured, the specifier can go away and via OPERATOR systematically produce the state schema, and the operation schemas as indicated above, bringing to bear all the power and formality of the mathematically based Z notation. Any revisions as a result of applying OPERATOR can be illustrated graphically and discussed with the client. All that remains now is to animate the Z specification that has been developed to enable the client to see, this time, the system in action. Animation is the subject of the next three chapters. Further discussion of OPERATOR is given in chapter 8.

REFERENCES IN CHAPTER 4

Sem91. Semmens, L. and Allen, P., *Using Yourdon and Z: an approach to formal specification*, Proceedings of the Fifth Annual Z User Meeting, Nicholls, J.E., Ed., Springer-Verlag, London, 1991.

Ran91. Randell, G., *Data Flow Diagrams and Z*, Proceedings of the fith annual Z User Meeting, Nicholls, J.E., Ed., Springer-Verlag, London, 1991.

Pol93. Polack, F., Whiston, M., and Mander, K., The SAZ Project: Integrating SSADM and Z, in FME '93 - Industrial Strength Formal Methods, Lecture Notes in Computer Science, Woodcock, J.C.P and Larson, P.G., Eds., Springer-Verlag, London, 1993.

Sul93. Sully, P., *Modelling the World with Objects*, 2nd edition, Prentice-Hall International, 1993.

Gra91. Gravell, A., M., *What is a good formal specification?*, in proceedings of the Fifth Annual Z User Meeting, Nicholls, J.E., Ed., Springer-Verlag, London, 1991.

Spi92. Spivey, J. M., *The Z Notation: A Reference Manual*, 2nd Edition, Prentice Hall International, 1992.

Coo92. Cooper, D., Mardell, J., Meehan, A., Norcliffe, A. and Valentine, S., *A Z Readers Course*, Shefield Hallam University Pavic Publications in Assn. with the D.T.I., 1992.

Nor91. Norcliffe, A. and Slater, G., *Mathematics of Software Construction*, Ellis Horwood Ltd., 1991.

Nor93. Norcliffe, A., *Computer Aided Modelling*, in Proceedings of the Eighteenth Undergraduate Mathematics Teaching Conference, Yardley, P. (Ed.), Shell Centre Publications, 1993.

CHAPTER 5: ANIMATING Z SPECIFICATIONS

MORIARTY. How are you at mathematics?
HARRY SECOMBE. I speak it like a native.
Spike Milligan, The Goon Show.

5.1 Introduction

The previous chapter was devoted to the issues concerning writing Z specifications but now we turn to those involved in animating them. Here we review some of the tool support that is currently being offered for Z and identify the need for tools that enable Z specifications to be demonstrated in a way that a typical software system client or user might understand. We discuss the advantages and limitations of animation and look specifically at how Z specifications might be animated using Prolog.

5.2 Computer aided Z.

The importance of tool support for contemporary software engineering has been discussed in previous chapters. Formal specification is an area ripe for exploitation in the development of computer based aids. Dedicated document or word processing systems for the mathematical notations and specification constructs are needed. The logical nature of the mathematics used urges the development of tools to automate the laborious process of proof required to formally verify specifications and to show that they are internally consistent. Tools to check grammar and syntax, the correctness of specifications, would be of immense benefit. Rapid prototyping of specifications, executing or animating specifications, must have a role to play. As Sommerville [Som92] puts it

"Formal specifications may be automatically processed. Software tools can be built to assist with their development, understanding and debugging. Depending on the formal specification language used, it may be possible to animate a formal specification to provide a prototype system."

There are other considerations that are dependent on the specification language used. The Z notation, for example, requires that each variable, each item of data, has a strictly specified type and that types cannot be mixed. Z specifications are structured, modularised, using schemas in which only specified variables are in scope. Tools to check for type miss-matches and for variables referred, to but out of scope, will be of great help to the specifier.

It comes with little surprise, therefore, that there has been and still is a great deal of activity and interest within this area of tool support for the popular Z notation, although most developments are still prototypical, very few being commercial, industry standard products. These are early days.

Most of the tools that are now available take the form of dedicated word processing systems with intelligent features like checking for correct syntax, grammar, variable type and variable scope, thus providing a means of producing printed and 'proof read' specification documents. These tools allow specifiers to work in a word processing style rather than on paper, and offer various means of indexing elements of a specification, expanding schemas to show hidden or included information, and for manipulating specifications. Clearly, to be able to show such intelligence as described above, such Z processors must be able to parse the notation or require some form of translation from the notation into a form able to be understood by the tool.

Probably the first example of a Z processor was the FORSITE Evaluation System [FOR87] which allows users to enter, edit, print and check the syntax and correctness of specifications written in Z. The emphasis of the system is on allowing the specifier to work on a specification in a printable form rather than using transformations and keywords, thus requiring a multiple font editor, upon which the system is heavily dependent. fUZZ [Spi88a] is a package offering similar facilities to

FORSITE - it allows printing of Z specifications and checks them for compliance with the Z scope and type rules - but takes a different approach in that the specification is not input or presented on screen in printable form. A specification has to be translated into a form to be processed by the LaTeX text formatting program [Lam86]. Natural language text can be entered as read but fUZZ defines ASCII keywords, prefixed by "\", to express Z constructs and symbols.

Formaliser [For90] is another tool of the same genre. Produced by Logica it builds on the pioneering tools described above by combining the input of printable form Z and parsing and printing of the fUZZ/LaTeX package. As well as providing facilities for editing and viewing, type and scope checking, it also allows interactive queries of attributes at points throughout a specification, such as displaying all the variables that can be referred to within a particular expression or showing the type of an expression. In addition, specification documents are held within a library allowing new documents to be created, existing documents to be copied, renamed, removed, and opened for editing. More than one document can be opened at one time and can be linked together allowing large specifications to be partitioned into convenient sections which can be edited and checked separately. We have a prototype Formaliser at Sheffield Hallam which has been used by staff and students producing excellent results.

A tool that is showing commercial success is CADiZ, produced by York Software Engineering at the University of York [Jor91]. Offering similar facilities to those described above, it operates within the UNIX environment rather than being PC based. It has sophisticated diagnostics of errors in Z syntax and a user-friendly specification browser including on-screen expansion of shcema calculus expressions. York Software Engineering are developing the tool to include aspects of verification, or specification proof, taking the tool far beyond the basic word processor level.

The Genesis Z Tool [Ash92] is another that, whilst also offering extensive checking and document handling facilities, moves into the realm of mathematical verification, allowing users to prove the validity of assertions made in their specifications via its 'extensible tactical proof system'.

Nevertheless, the tools described above are concerned only with the production of printable Z specifications that have been automatically checked for errors in Z syntax and grammar. They do not purport to demonstrate system behaviour. Tools that automate mathematical proof, such as CADiZ, the Genesis Z Tool and dedicated Z proof tools such as zedB [Nei91], are invaluable for showing that, for example, a system state can exist, or that a correct precondition for a change of state has been specified, or that a system operation does not violate the specified constraints on the system state. They cannot, however, prove semantic properties of a specification. They cannot execute a specification to show what the system actually does.

5.3 Animation - advantages and limitations.

Producing system prototypes has been a significant feature of software engineering for some time now, and so-called rapid prototyping, where a client is presented with a working model of a software system in the fastest possible time, is a popular practice. Whilst the value of a prototype is well understood in traditional engineering disciplines it is useful to put it in software engineering terms - take Blum's definition [Blu92], for example:

"A prototype software is a partially complete functional model of a target system. Its purpose is to provide a better understanding of the target system's requirements."

Prototypes are cheaper than implementations; they need only model a system and are not encumbered with expensive peripherals. They give the opportunity to 'test before you buy' and provide a medium for communication between provider and consumer. Kenmore Braithwaite [Bra90] makes several important points:

"The major assumption underlying the introduction of prototyping is that software development is an interactive design process. Effective design is achieved only as a result of feedback between designer and user... The prototype is designed with the expectation of change...Prototyping tools play an important part in automating the early life-cycle phases. They are used to determine system requirements and answer questions about the behaviour of the emerging system...One demonstration is better than two volumes of specifications."

Animation, in essence, is rapid prototyping applied to formal specification. Animation refers to the production of a working model of a formal specification that retains, as far as is possible, the characteristics of mathematical rigour and abstraction. An animation of a formal specification demonstrates the essential features and behaviour of a system whilst remaining faithful to and consistent with the mathematical model.

Typically, and until some sort of parsing of formal specifications to produce computer generated animations is achieved, the production of an animation will require some form of translation from the mathematical notation to an executable language without any need, or indeed scope, for subjective understanding. As Dick et al [Dic90] put it:

"Having invested in a formal specification, it is highly desirable that the process of interpretation should have a formal basis. Thus, in order that we can say that an animation is a formal specification, the transformations must remain faithful to the structure and semantics of the formal notation used"

An animation, then, may be thought of as an executable formal specification, although, by its nature, must be less abstract than a mathematical model can be. If we ignore the usefulness of a user interface we must still consider concrete items of data, for example. Hayes and Jones [Hay89] point out other limitations of the animation approach:

"Many formal constructs are not easily transformed into code, especially non-deterministic expressions and some deterministic expressions when quantifiers (∀, ∃) are used."

Another consequence of using animation to demonstrate formal specifications is that the developer is likely to be restricted to a subset of a formal notation. After all, mathematics, free from computational constriction, is boundless in its power of expression. Hayes and Jones believe that such restrictions are undesirable, and they clearly are, but they are also unavoidable. So why throw the baby out with the bath water?

Referring to Braithwaite again:

"Executable specification languages are the most sophisticated prototyping tools. They change system development into an interactive process where the system is specified and the specifications are executed to determine if the system is complete and correct. Then, based on the experience of using the prototype, the specifications are refined and then re-executed. This interactive process continues until the system is able to perform in a manner that meets all the user requirements."

In summary, then, (and see Fig. 5.1) animation enables the software developer to

- validate a formal specification by demonstrating it to a client or user
- reason with a specification with respect to system behaviour

In certain cases, an animation may even be suitable as an implementation.

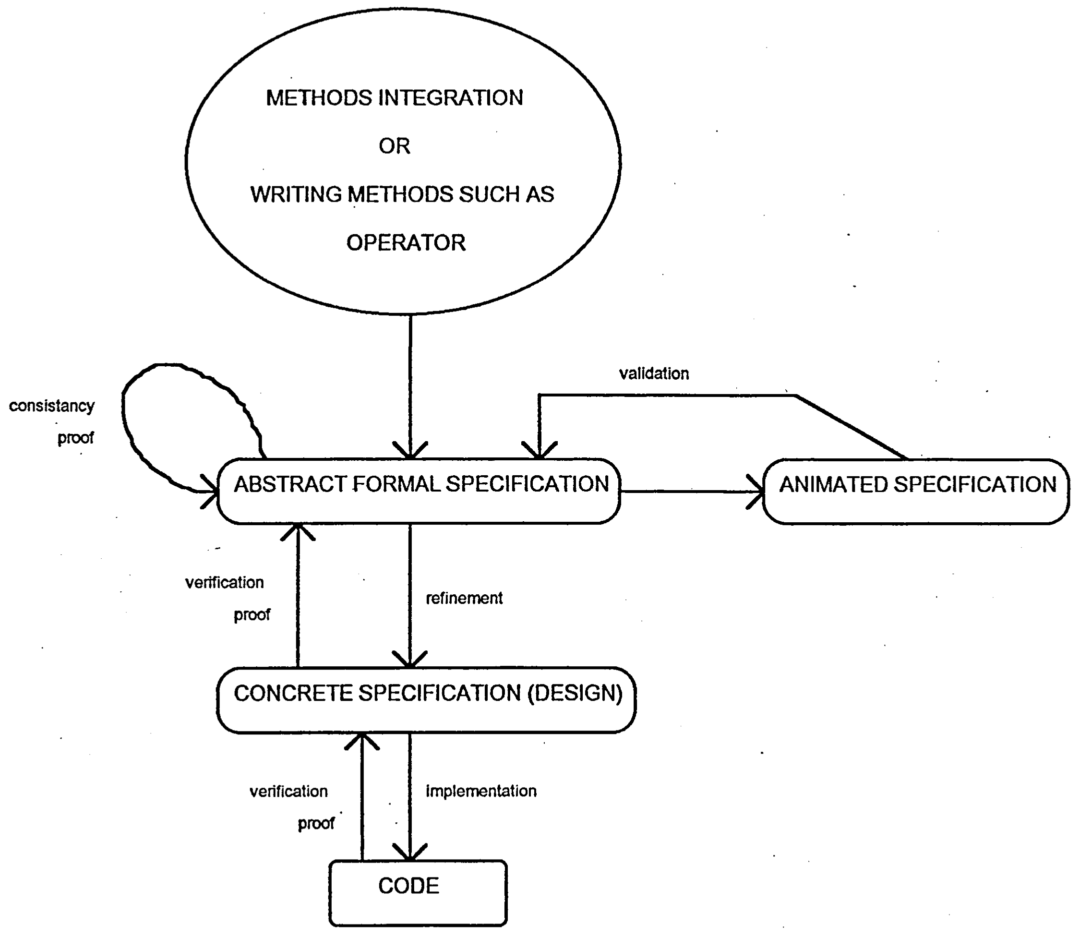


Fig. 5.1: Animation as part of a formal software engineering process.

5.4 Animation Techniques for Z.

The basic process of animation is of translating the mathematical specification into an executable form whilst preserving its structure and the grammar and semantics of the mathematical language. For example, if the mathematical specification describes the union of two sets then the animation should describe the union of two sets.

In the case of animating Z this usually involves implementing or simulating set theory, functions, relations, sequences etc., as well as retaining as far as possible schemas and the schema calculus (for error handling, for example), the constraints on the system state (data invariants) and the strict data typing of Z. For the purpose of validation with a client or user it is also desirable to have a user-friendly interface to the animation and, possibly, other means of demonstration. All the techniques of animation we have seen have used Spivey [Spi92] as a standard text for Z.

Animation is often a process of straightforward translation into a target language using a set of pre-defined rules of translation and , possibly, a library of pre-programmed Z operations. In its most sophisticated form, animation might be carried out in a systematic way using an 'intelligent' animation tool, or animator. One interesting alternative approach has been suggested by Sam Valentine [Val92] who has produced an executable subset of Z, *Z--*.

There are many programming languages and environments that are suitable targets for the purpose of animating Z and especially popular are the so-called fourth generation languages that support logical expressions. Success has been had, for example, Diller, using Miranda [Dil90], a functional programming language, by Morrey et al. [Mor90], using Lisp, a predicate based list processing language, and Love, using SQL forms [Lov93].

Perhaps the greatest interest has been shown in using Prolog as a medium for animating Z. The achievements of Dick, Cozens and Krause (based on pioneering work by Ron Knott at the University of Surrey) are worth particular note with their development of a Z-to-Prolog translator, animator and transformation system [Dick90]. Also see [Kno91].

This widespread interest in Prolog gave us an obvious starting point for looking at the issues of animation.

There is good correlation between Z and Prolog, both are based on predicate logic and are declarative in nature. There are few procedural considerations in Prolog and Prolog predicates and clauses have many similarities with schemas in Z. The domains section of some versions of Prolog, such as Turbo Prolog [Pro86] allow the construction of named types, while databases are ideal for simulating sets and most versions support lists (sequences in Z).

Take, for example, the simple security system specification, given in *A Z Readers Course* [Coo92]. Members of staff in an organisation are represented by three sets of unique identification codes. The members of staff in the organisation's building are represented by *in*, those out of the building by *out*, and the overall staff membership by *users*. The state schema is:

State

$in, out, users : \mathbb{P}\ STAFF\_ID$

$in \cap out = \varnothing$
$in \cup out = users$

In Turbo Prolog the parachuted type and the state variables (along with an additional variable to be used later for input) are created thus:

```
domains
      staff_id = symbol
database
      in(staff_id)
      out(staff_id)
      users(staff_id)
      input(staff_id)
```

The operation to check a member of staff into the building is specified as:

CheckIn

$\Delta State$

$staff? : STAFF\_ID$

$staff? \in out$

$in' = in \cup \{ staff? \}$

$out' = out \setminus \{ staff? \}$

$users' = users$

When the precondition, $staff? \in out$, is violated the *CheckIn* operation will fail. A robust *CheckIn* operation, *RCheckIn*, can be be specified as follows:

$RCheckIn \mathrel{\widehat=} (CheckIn \wedge Success) \vee CheckInError$ where

$CheckInError \mathrel{\widehat=} StaffIn \vee NotUser$

The success and error schemas being:

Success

$result! : REPORT$

$result! = ok$

StaffIn

ΞState
staff? : STAFF_ID
result! : REPORT

---

staff? $\in$ *in*
result! = staff_in

NotUser

ΞState
staff? : STAFF_ID
result! : REPORT

---

staff? $\notin$ *users*
result! = not_known

In Turbo Prolog we use the commands `assert` and `retract` to add and remove elements to and from sets, and `readln` for input. The individual predicates provide the schema calculus while the clauses by which each predicate is defined represent the Z text:

```
rcheckIn :- checkIn, success.
rcheckIn :- checkInError.
checkInError :- staffIn.
checkInError :- notUser.

checkIn :- write("enter name"), nl,
      readln(X), assert(input(X)), out(X),
      assert(in(X)), retract(out(X)),
      retract(input(X)).

success :- write("ok"), nl,

staffIn :- input(X), in(X), write("staff in"), nl,
      retract(input(X)).

noutUser :- input(X), not(users(X)),
      write("not known"), nl,
      retract(input(X)).
```

Turbo Prolog databases are used in a similar fashion to represent Z functions:

$function' = function \oplus \{ x? \mapsto new? \}$ in Z, for example, becomes

```
        retract(function(X, _)),
        assert(function(X, New)).
```
in Turbo Prolog.

Sequences are well catered for in Prolog using lists:

$sequence : \mathrm{seq}\ TYPE$
$x : TYPE$

$x = \mathrm{head}(sequence)$

in Z, can be modelled in Turbo Prolog, using the list constructor, `*`, by:

```
domains
        x = symbol
        sequence = x*

predicates
        head_of_sequence(x, sequence)

clauses
        head_of_sequence(X, [Head | Tail]) :-
                X = Head.
```

So, clearly, Prolog has many features that make it an excellent vehicle for animating Z, although the version of Prolog used here, Turbo Prolog, is a limited version of the full standard Prolog and we do not present the more sophisticated approach of [Kno91], for example. However, Prolog does have drawbacks in that it can be inefficient - in our experience some lengthy animations have been ponderously slow. It is also clear that a deal of effort would be required to produce an animator that had the desired user-friendly interface to make it practical for demonstration to a client or user. These drawbacks, and the fact that we felt it more useful to investigate more novel approaches to animation, indicated to us that we might look towards some existing environment or shell that could provide tool-based support and pre-existing interface facilities. One possibility that presented itself to us was the expert system shell, Crystal.

REFERENCES IN CHAPTER 5

Som92. Sommerfield, I., *Software Engineering*, 4th Edition, Addison-Wesley, 1992.

FOR87. *FORSITE Evaluation System*, User's Guide, Alvey FORSITE Project, Sept. 1987.

Spi88a. Spivey, J., M., *The fUZZ Manual*, J.M.Spivey Computing Science Consultancy, Oxford, 1988.

Lam86. Lamport, L., *LaTeX: A Document Preparation System*, Addison-Wesley, 1986.

For90. *Formaliser User Guide (Z Specification Release)*, Version 6.0, Logica Cambridge Ltd., Nov., 1990.

Jor91. Jordan, D., McDermid, J., A. and Toyn, I., *CADiZ - Computer Aided Design in Z*, proceedings of the Fifth Annual Z User Meeting, Oxford 1990, Springer-Verlag, 1991.

Ash92. Ashoo, K., *The Genesis Z Tool - An Overview*, FACS Facts, The Newsletter of the BCS Formal Aspects of Computing Science SIG, May, 1992.

Nei91. Neilson, D. and Prasad, D., *zedB: A proof tool for Z built on B*, proceedings of the Sixth Annual Z User Meeting, University of York, 1991.

Blu92. Blum, B., I., *Rapid Prototyping of Information Management Systems*, ACM-SIGSOFT Software Engineering Notes, Vol. 7, No. 5, Dec., 1992.

Bra90. Braithwaite, K., S., *Applications Development Using CASE Tools*, Academic Press Incorporated, 1990.

Dic90. Dick, A., Krause, P. and Cozens, J., *Computer aided transformation of Z into Prolog*, in *Z User Workshop*, J.E.Nicholls, editor, Workshops in Computing, Springer-Verlag., 1990.

Hay89. Hayes, I., J. and Jones, C., B., *Specifications are not (Necessarily) Executable*, Software Engineering Journal, Nov., 1989.

Spi92. Spivey, J. M., *The Z Notation: A Reference Manual*, 2nd Edition, Prentice Hall International, 1992.

Val91. Valentine, S., *Z--, an Executable Subset of Z*, in *Z User Workshop*, York 1991, Nicholls, J.E., Ed., Springer-Verlag, 1992.

Dil90. Diller, A., *Z: An introduction to Formal Methods*, John Wiley, 1990.

Mor90. Morrey, I., Siddiqi, J., and Shaw, *Rapid Prototyping of Formal Specifications*, Sheffield Hallam University, to be published, 1990.

Lov93. Love, M., *Animating Z specifications in SQL*Forms3.0*, in *Z User Workshop, London 1992*, Bowen, J.P. and Nicholls, J.E., Eds., Workshops in Computing, Springer-Verlag, 1993.

Kno91. Knott, R., D., *Making Discrete Mathematics Executable on a Computer*, in *The Mathematical Revolution Inspired by Computing*, Johnson, J.H. and Loomes, M.J (Eds.), The Institute of Mathematics and its Applications Conference Series, New Series No. 30, Clarendon Press, Oxford, 1991.

Pro86. *Turbo Prolog, the Natural Language of Artificial Intelligence*, Borland International Inc., 1986.

Coo92. Cooper, D., Mardell, J., Meehan, A., Norcliffe, A. and Valentine, S., *A Z Readers Course*, Sheffield Hallam University Pavic Publications in Assn. with the D.T.I., 1992.

CHAPTER 6: ANIMATING Z SPECIFICATIONS - CRYSTAL

Mathematics may be defined as the subject
in which we never know what we are talking about,
nor whether what we are talking about is true.
Bertrand Russell, Mysticism and Logic.

6.1 What Crystal has to offer

Crystal is reasonably well known in artificial intelligence circles and is sold as an expert system shell by Intelligent Environments Ltd. in Richmond, London [Cry87].

It is a rule-based programming language offering excellent input, output and menu creation facilities, as well as all the standard features expected of any expert system shell, such as forward and backward chaining, and reporting on the success or failure of rules via its Rule Trace system.

A typical rule in Crystal might well be the following (here written at the highest level before any low-level coding constructs are considered):

```
            Rule works
IF          A is true
AND NOT     B is true
OR          C is true
```

By refining the subrules, for example, `A is true` is a subrule, by eventually involving the low-level constructs of Crystal, the rule is thereby implemented.

The specific advantages that this environment offered as a means of animating Z were perceived to be as follows, see also [And90, And91]:

- The rule-based nature of Crystal means that lines of Z, in the predicate of a schema, transform almost one-for-one into rules in Crystal.

- The expandable way in which rules are built up in Crystal mirrors very closely the use of the schema calculus in Z. The developer, using the tool, can faithfully transform a Z specification starting at the schema level and finishing at the line-by-line predicate level.

- The user interface builder that comes with Crystal, including facilites for creating menus and display forms, and input and output fields, enables the developer to concentrate his efforts on transforming Z instead of worrying about how to create a user friendly interface. This is an added bonus given the fact that implementation issues are positively avoided in formal specifications.

- The animation that results can be viewed by the client at different levels. This is possible because of the folded nature of the rule-based programming in Crystal. At the highest level a system might be viewed as a menu having several options such as

```
quit
initialise state
save state
load state
print state
test data invariants
operation 1
operation 2
:
:
operation n
```

- Any operation chosen by the client can be systematically unfolded to discover the rules that make it work, thus promoting the vital interaction between client, developer and system that is necessary for requirements validation. In Crystal this is feasible because at the highest level the rules are written in English. Only at the lowest level does English give way to code. What the client sees, therefore, is a faithful English translation of the developer's Z.

6.2 Using Crystal for animation

To illustrate these points a short example is now considered. The following is part of the Z specification of a very simple security system that might be in operation in a building to monitor the whereabouts of staff users. The system is taken directly from *A Z Readers Course* [Coo92], and is identical to the example used in the previous chapter. As before, the system state consists of three subsets, *in*, *out*, and *users*, of type $\mathbb{P} STAFF\_ID$, and is represented by the following state schema:

State

$in, out, users : \mathbb{P}\ STAFF\_ID$

$in \cap out = \varnothing$
$in \cup out = users$

Amongst other things, and again as before, the system checks people in to the building. For completeness, we show again how the robust *CheckIn* operation is arrived at starting with the *CheckIn* operation schema:

CheckIn

$\Delta State$
$staff? : STAFF\_ID$

$staff? \in out$
$in' = in \cup \{ staff? \}$
$out' = out \setminus \{ staff? \}$
$users' = users$

When the precondition, *staff?* $\in$ *out*, is violated the *CheckIn* operation will fail. A robust *CheckIn* operation, *RCheckIn*, can be be specified as follows:

$RCheckIn \mathrel{\widehat{=}} (CheckIn \wedge Success) \vee CheckInError$ where
$CheckInError \mathrel{\widehat{=}} StaffIn \vee NotUser$

The success and error schemas being:

```
__ Success ____________________________________
|
|  result! : REPORT
|______________________________
|
|  result! = ok
|______________________________________________
```

```
__ StaffIn ____________________________________
|
|  ΞState
|  staff? : STAFF_ID
|  result! : REPORT
|______________________________
|
|  staff? ∈ in
|  result! = staff_in
|______________________________________________
```

NotUser

$\Xi State$

$staff? : STAFF\_ID$

$result! : REPORT$

$staff? \notin users$

$result! = not\_known$

At the highest level the Crystal coding for this Z could be the following:

```
      RCheckIn works
IF    CheckIn works
AND   Success is indicated
OR    CheckInError works

      CheckInError works
IF    StaffIn applies
OR    NotUser applies
```

At the next level down these rules might be expanded as follows:

```
      CheckIn works
IF    staff_id is entered into the system
AND   the staff_id currently belongs to the set out
AND   the staff_id is then removed from the set out
AND   the staff_id is then added to the set in
AND   the set users is unchanged

      Success is indicated
IF    the result "ok" is output

      StaffIn applies
IF    staff_id is entered into the system
AND   the staff_id currently belongs to the set in
AND   the result "staff in" is output

      NotUser applies
IF    staff_id is entered into the system
AND   the staff_id does not currently belong to the set users
AND   the result "not known" is output
```

Obviously, the developer has to expand each of these individual rules further until they are capable of being executed. But, in principle, this is a fairly straightforward task given the available Crystal commands, and the fact that sets, functions, relations, sequences, power sets, bags, etc. can all be represented conveniently as arrays in Crystal.

Before looking at the advantages of this approach to animation it will be useful to look at another more involved example, where we include some lower level coding.

The example we consider is one of a library system. Note that this is not the same system as that described in Chapter 4, although the requirements are very similar:

"A library has members of staff and borrowers of books. Staff can also borrow books. The library records the borrowers of copies of books and keeps a database of book details. There is a limit to the number of books any one person can borrow."

The library state schema we arrive at is:

LibState

$staff : \mathbb{P}\ PERSON$
$borrowers : \mathbb{P}\ PERSON$
$books\_in : \mathbb{P}\ COPYID$
$checked\_out : COPYID \nrightarrow PERSON$
$book\_db : COPYID \nrightarrow BOOK$
$max\_books : \mathbb{N}$

$staff \cap borrowers = \emptyset$
$books\_in \cap \mathrm{dom}\ checked\_out = \emptyset$
$books\_in \cup \mathrm{dom}\ checked\_out = \mathrm{dom}\ book\_db$
$\mathrm{ran}\ checked\_out \subseteq staff \cup borrowers$
$\forall p : \mathrm{ran}\ checked\_out \bullet \#\ checked\_out \rhd \{\ p\ \} \leq max\_books$

In Crystal the state variables *staff*, *borrowers* and *books_in* are represented by one-dimensional arrays. The variables *check_out*, and *book_db* are represented by two-dimensional arrays. Arrays are created in Crystal before coding commences. Single element variables such as *max_books* are created within the Crystal code when they are first assigned their values. All variables are global within Crystal within a given application.

Data variant testing is provided by the Crystal rule `Test-lib-data` shown below:

```
Test-lib-data
IF Assign m := 0
      AND No-staff-is-ordinary-borrower-and-vice-versa
      AND Assign m := 0
      AND No-copy-can-be-in-the-lib-and-checked-out
      AND Assign m := 0
      AND All-lib-copies-have-book-info
      AND Copies-are-checked-out-to-staff-or-borrowers
      AND No-borrower-can-have-more-than-maxbooks-out
```

Note that each line in the state schema predicate has become a line of English text, each being another Crystal rule. The quantity `m` is simply an array counter, assigned the value zero before array searches are carried out.

Each of the lines of English text, i.e. each Crystal rule now has to be expanded. For example, the No-staff-is-ordinary-borrower-and-vice-versa becomes:

```
No-staff-is-ordinary-borrower-and-vice-versa
IF Test staff$[m] = "empty"

OR Assign id$ := staff$[m]
      AND Assign n := 0
      AND Search-for-id-in-borrowers
      AND Test tf = 0
      AND Assign m := m + 1
      AND Restart Rule
```

The `search_for_id_in_borrowers` rule is finally expanded in terms of Crystal code as follows:

```
Search-for-id-in-borrowers
IF Test borrowers$[n] = id$
      AND Assign tf := 1

OR Test borrowers$[n] = "empty"
      AND Assign tf := 0

OR Assign n := n + 1
      AND Restart Rule
```

The above describes an array search whereby if a match is found a test flag, `tf`, is set to 1, causing the rule `No-staff-is-ordinary-borrower-and-vice-versa` to fail.

The rules `No-copy-can-be-in-the-lib-and-checked-out`, `All-lib-copies-have-book-info`, etc. can be similarly expanded and implemented.

The library system operation of checking out a book to a borrower can be specified using schema calculus as follows:

$$
\begin{aligned}
\mathit{RCheckOutBook} \mathrel{\widehat{=}} \ & \mathit{CheckOutBook} \lor \\
& \mathit{NotBorrower} \lor \\
& \mathit{BookNotIn} \lor \\
& \mathit{TooManyBooks}
\end{aligned}
$$

where the *CheckOutBook* operation is the following schema:

CheckOutBook

$\Delta LibState$

$borrower? : PERSON$

$copy? : COPYID$

$report! : REPORT$

$borrower? \in borrowers \cup staff$

$copy? \in books\_in$

$\# \, checked\_out \rhd \{ borrower? \} < max\_books$

$checked\_out' = checked\_out \cup \{ copy? \mapsto borrower? \}$

$books\_in' = books\_in \setminus \{ copy? \}$

$report! =$ "book checked out"

$staff' = staff$

$borrowers' = borrowers$

$book\_db' = book\_db$

$max\_books' = max\_books$

And *NotBorrower*, *BookNotIn* and *TooManyBooks* are the associated error schemas, not given here for simplicity. The Crystal animation begins with the following `RCheck-out-book` rule:

```
      RCheck-out-book
IF    Check-out-book
OR    Not-borrower
OR    Book-not-in
OR    Too-many-books
```

Here, each subrule represents a schema and is systematically expanded. `Check-out-book`, for example, becomes:

```
Check-out-book
IF            Get-borower-id
      AND Get-copy-id
      AND Borrower-id-is-in-staff-or-borrowers
      AND Copy-is-in-library
      AND Borrower-doesnt-have-too-many-books-out
      AND Put-borrower-and-copy-in-checkedout
      AND Delete-copy-id-from-booksin
      AND Display Form

                    | BOOK CHECKED OUT |
```

Note the use of a Crystal Display Form to provide output.

`Get-borrower-id` is simply another Display Form, this time with an input field:

```
Get-borrower-id
IF    Display Form

                    | Enter The Following:      |
                    | The borrower id  < id$ >  |
```

The expansion of `Put-borrower-and-copy-in-checkedout` demonstrates the animation of a postcondition:

```
Put-borrower-and-copy-in-checkedout
IF    Assign n := 0
      AND Find-end-of-checkedout
      AND Assign checkedout$[n,0]  := cid$
      AND Assign checkedout$[n,1]  := id$

Find-end-of-checked-out
IF    Test checkedout$[n,o]  =  "empty"

OR    Assign n := n + 1
    AND  Restart Rule
```

In this way, using the approaches described above, the other predicates of *CheckOutBook*, the error schemas associated with it, and, eventually, the entire system specification for the library could be animated.

6.3 Use in the classroom

The use of Crystal to animate Z has been used in the teaching of formal methods on the MSc Engineering Information Technology course at Sheffield Hallam University. The author has used the approaches outlined above for three years in the teaching of animation and in tutorial work on animation. Animation, using Crystal, has been chosen by groups of students as part if their learning contracts (as discussed in Chapter 3). This has resulted in successful animations of several systems - a notable one being the intelligent multi-storey car parking system. Animation of Z specifications using Crystal has also been the subject of MSc projects, supervised by the author.

6.4 Advantages and limitations

Some of the advantages of using Crystal to animate Z specifications have been listed previously in 6.1, but it is worth expanding on these:

- We observe that the Crystal is very faithful to the Z. The simulation that is produced when the Crystal code is executed is indeed an animation of the specification and not an implementation that is far removed from Z.

- Since the Crystal mirrors the structure of the Z closely it is a relatively easy task for the developer to begin the process of developing the executable code. The developer takes the specification schema by schema, line by line, to arrive at the animation.

- There is a high degree of reusability of Crystal rules. Rules representing error schemas can be referenced again, in the same way as is common in error handling in Z. In addition, rules representing individual lines of Z are also reusable.

- The high level coding, being written in English, is clearly capable of being understood by a client even though he may know little or no Z. The English translation of the Z in Crystal does not introduce potentially harmful ambiguities and via this translation the client can thus interact with the specification and contribute meaningfully to the process of requirements validation.

- The three-way communication between customer, developer, and system, so vital for validation purposes, is thus possible via this approach.

The Crystal approach does have its limitations. A major disadvantage of Crystal is that although at a high level it faithfully represents the Z notation, at the lowest level the Crystal code can be somewhat lengthy. For example, the Crystal transformation of the function override operation could require upwards of 50 lines of coding. This problem is compounded by the fact that there is no parameter passing in Crystal, all variables being global; it is not possible to write a single routine for function override, for example, and pass the appropriate parameters to it. The code must be repeated each time it is required with the new variable names inserted in the rules.

Another disadvantage is that, in developing an animation, it is not possible to incorporate the strict data typing feature of the Z notation.

These limitations, of lengthy code, lack of parameter passing, and the inability to incorporate type checking were sufficient to persuade the author that Crystal did not readily possess the features needed to implement Spivey's Mathematical Tool-kit for Z [Spi92]. Intelligent Environments assured us that interfaces to Crystal written in C could be produced to surmount the parameter passing problem - and therefore to implement a library of Z operations. However, in the end, it was decided to turn to the more sophisticated Windows-based environment, Kappa PC, which offered excellent interface facilities as well as parameter passing and therefore the possibility of implementing the Mathematical Tool-kit of Z.

REFERENCES IN CHAPTER 6

Cry87. *Crystal: The Expert System Builder*, Users Guide, Intelligent Environments Ltd., Richmond, London, 1987.

And90. Andrews, S., J. and Norcliffe, A., *A CASE Tool for Demonstrating Z Specifications*, IEE Colloquium Digest No. 1990/058, April, 1990.

And91. Andrews, S., J. and Norcliffe, A., *An Expert System CASE Tool for Simulating Z Specifications*, Polymodel 13 Conference Proceedings, 1991.

Coo92. Cooper, D., Mardell, J., Meehan, A., Norcliffe, A. and Valentine, S., *A Z Readers Course*, Sheffield Hallam University Pavic Publications in Assn. with the D.T.I., 1992.

Spi92. Spivey, J. M., *The Z Notation: A Reference Manual*, 2nd Edition, Prentice Hall International, 1992.

CHAPTER 7: ANIMATING Z SPECIFICATIONS - ZAPPA

Just like a penguin in bondage, boing!
Lyric by Frank Zappa, Artiste (deceased).

7.1 Introduction.

In chapter 6 we described an approach to animation using the expert system shell Crystal. One of the disadvantages of Crystal as an animation medium was that it was not possible to build a library of functions to represent the Mathematical Tool-kit of Z - the Tool-kit as given by Spivey [Spi92] - due to the lack of parameter passing in Crystal. Whilst not being a necessity, such a library is desirable if serious animation work is to be carried out, since such a library would considerably reduce the amount of code required to animate a Z specification.

One particular advantage of Crystal, however, was that a user-friendly interface was easy to create. A user-friendly interface to an animation was deemed to be an important feature, and its ease of creation is obviously of concern to the developer.

Ideally, we wished to combine a library of Z operations with an excellent user interface, and we felt that this might be taken further by providing the developer with an interface for the purpose of building an animation in a systematic fashion. In addition, if the Mathematical Tool-kit of Z were to be implemented then, ideally, type and syntax checking would have to be provided. Given that the construction of animation tools had been identified as a key issue in the demonstration of Z specifications, the idea of building an animator CASE tool began to take shape.

What was needed was an environment that provided excellent interfacing facilities, perhaps Windows-based, with a logic-based programming language whereby Z's Mathematical Tool-kit might be implemented.

At the time this need was identified, the Kappa-PC environment [Kap90] had just been acquired by Sheffield Hallam University (for the Schools of EIT, Engineering and Computer Management Sciences). Upon inspection and the advice of experienced users it was decided to investigate the possibility of constructing an animator using Kappa-PC.

In this chapter we therefore describe the features of Kappa-PC that we felt to be important for the creation of an animator and give an overview of the prototype tool, Zappa, that was eventually created. We then describe the use of the tool with the aid of a simple example before looking in detail at how the implementation of Z data structures (and, consequently, the Z Tool-kit library) was achieved. We go on to explain how Zappa was used by students and conclude by making an evaluation of the tool.

7.2 What Kappa has to offer.

Kappa-PC is a knowledge-based systems builder, for the mouse driven Windows environment, produced by InteliCorp Inc. It has several features that indicated that it might be a suitable vehicle for the development of an animation tool:

- It supports its own logic-based programming language supporting parameter passing, KAL (Kappa Applications Language), that offered the possibility of creating a Z operation library.

- It is an object oriented environment in which objects are easily created, given properties, manipulated and examined. Z structures such as schemas and state variables might be conveniently viewed as objects.

- Multiple session windows allow systems to be subdivided using separate interfaces. Thus it would be possible to have separate session windows for an animator and an animation. In fact, in Kappa-PC it is possible to create one application that can then be used to create another.

- Buttons can be created within session windows to help provide a user friendly interface.

- There are sophisticated Windows-based interface features such as pop-up menus for single and multiple selection, for selecting a basic type from a list of basic types, for example, and there are various other forms of input and output that might facilitate the animation of operation schemas or investigations of the system state.

- There is also a range of editing facilities and an error tracing mechanism.

7.3 An Overview of Zappa.

The first thing to note is that Zappa can only animate specifications which conform to the conventions of procedural systems as given by Spivey [Spi92]. The specifications also have to be deterministic with all schema output variables, preconditions and postconditions given explicit definitions. After-state variables and output variables must appear on the left hand side of predicates that define their values. The Z written so far in this thesis has been writen in this way.

Zappa uses two Kappa-PC session windows. The first is the ANIMATOR screen which the developer uses to construct animations. The second is the ANIMATION screen, in which the interface to the animation is progressively formed. It is the intention of the tool that a would be client or user would use the ANIMATION screen to investigate essential features of the system that has been animated. The ANIMATION screen corresponds well with a typical animation menu screen created using Crystal for animation.

Kappa-PC's Object Browser gives, in essence, a third screen - showing an overview of the animated specification, displaying the names of schemas and the variables associated with them.

A fundamental difference between Zappa and the Crystal approach, however, is the idea of an animator. By pressing the various buttons of the ANIMATOR screen the developer accesses the features of the tool that aid in the creation of an animation. For example, there are buttons that access features for parachuting basic types into an animation and for creating state variables and schemas. Buttons are also provided for switching between the three screens. A complete list of ANIMATOR screen buttons is given in Fig. 7.1. A list of buttons that are always present in the ANIMATION screen (whether a specification animation is loaded or not) is given in Fig. 7.2.

Other key features of the tool include:

- A systematic and robust approach - the system will not allow the developer to create an operation schema until the state schema has been created, for example.

- The use of templates for the animation of schema predicates.

- Syntax and type checking.

- The ability to save and load animations.

OVERVIEW	ANIMATION
KTOOLS	NEW
Load Spec	Save Spec
QUIT	Show Variable
Create State Schema Box	Parachute Type
Make Free Type*	Make Schema Type*
Create State Variable	Create Data Invariant
Create Initial State	Create Input Variable
Create Output Variable	Create Local/Dummy Variable
Create Operation Schema	Create Robust Op Schema
Delete State	Delete Type
Del State Var	Delete DI
Delete Init	Delete Input
Delete Output	Delete Local
Delete Op	Delete Error
Delete ROp	

*Not yet implemented.

Fig. 7.1 ANIMATOR screen buttons.

overview
animator
ktools
show variable

Fig. 7.2: Permanent ANIMATION screen buttons

7.4 Using Zappa - an example.

To see the tool in action and to demonstrate many of its features we shall take the example of animating the banking system given in *A Z Readers Course* [Coo92].

The specification models accounts with overdraft facilities and is essentially the one used in chapter 4. We shall consider the animation of the state schema, an initial state (that of a bank with no accounts) and one robust operation, the operation to open an account.

The state schema *Account*, is given below:

$[ACC\_NO]$

Account

$balance : ACC\_NO \rightarrowtail\!\!\!\!\!+ \mathbb{Z}$
$od\_limit : ACC\_NO \rightarrowtail\!\!\!\!\!+ \mathbb{Z}$

$\mathrm{dom}\ balance = \mathrm{dom}\ od\_limit$
$\forall x : \mathrm{dom}\ balance \bullet balance\ x \geq od\_limit\ x$
$\forall x : \mathrm{dom}\ od\_limit \bullet od\_limit\ x \leq 0$

The first task is to parachute the basic type *ACC_NO* into the animation. This is achieved by pressing the `Parachute Type` button on the ANIMATOR screen. The user is then prompted to enter the name of the type. Typing errors are corrected using the usual backspace and arrow keys. Once entered, Zappa will tell the user that the type has been parachuted into the animation.

At this stage the user could check that *ACC_NO* had been parachuted into the animation by looking at the OVERVIEW. *ACC_NO* would be connected to the word `ParaTypes` in the OVERVIEW screen.

Next, the state schema, *Account*, is created. In Zappa, state schemas are created in three stages; the schema box, the state variables and the data invariant. The schema box is created by clicking on `Create State Schema Box` and entering the state schema name. The state variables are then created individually via the `Create State Variable` button. *balance* is entered as the first state variable and a pop-up menu appears, listing possible Z data structures such as tuple, function, sequence, etc. In this case the function option is selected.

Another pop-up menu is then used to select the form of the function from the list of forms supported by Zappa (a list of Z data structures currently supported by Zappa is given in figure 7.3). In this case $A \nrightarrow B$ is selected. A third menu appears, this time listing parachuted types along with $\mathbb{Z}$, $\mathbb{N}$ and $\mathbb{N}_1$, prompting the user to select the type for A. In this case *ACC_NO* is selected. A fourth menu prompts for the type for B, in this case $\mathbb{Z}$. Zappa will then tell the user that *balance* has been created, giving its declaration. The function *od_limit* is created similarly.

Again, the OVERVIEW screen could be consulted to show that *balance* and *od_limit* are associated with the state schema *Account*.

Note that when a function is created by the user, Zappa creates an algorithm to animate the function, giving it the property of functionality. This allows the function to be used in the form f(x) in animated predicates to provide output from the function.

A	𝔽 A
AxB	𝔽 (AxB)
AxBxC	𝔽 (AxBxC)
Ax(BxC)	𝔽 (Ax(BxC))
(AxB)xC	𝔽 ((AxB)xC)
seq B	A ⇸ B
seq (BxC)	A ⇸ (BxC)
	(AxB) ⇸ C

Figure 7.3: Z data structures currently supported by Zappa.

The next stage is to create the data invariant using a Zappa template. The `Create Data Invariant` button is clicked on and Zappa instructs the user how to proceed. A small window called KTOOLS appears (this is part of Kappa-PC). The user clicks on `Function` in the KTOOLS window, drags the arrow pointer down the pop-up menu that appears, to highlight `Edit`, and releases the mouse button. An `Edit Function` menu appears (again, part of Kappa-PC) and the user clicks on `DICheck` (standing for Data Invariant Check). A template (of KAL code) for the data invariant is displayed for the user to edit:

```
{
If NULL
      Then zmessage("data invariant ok")
      Else zmessage("data invariant error");
zend();
};
```

The user must now translate the data invariant of the state schema, *Account*, from Z to Zappa's Mathematical Tool-kit equivalent, and enter it where the `NULL` is in the template. The user clicks on the template, just after the `NULL` to position the flashing text editor cursor, and erases `NULL` using the backspace key.

The first predicate of the data invariant of *Account* is

dom *balance* = dom *od_limit*

In Zappa all Z toolkit functions begin with the letter Z and, as far as is possible conform to a direct or natural language translation of the Z notation. For example, dom in Z translates simply to `zdom` in Zappa.

In-fix functions and relations translate to post-fix form, hence $x = y$ in Z tanslates to `zequal?(x,y)` in Zappa, the question mark indicating that the function is a logical test returning true or false. A list of Zappa Z functions currently supported, along with the equivalent Z notation predicates, is given in figure 7.4.

So, using nesting of functions, the first predicate of the data invariant becomes, in Zappa:

```
zequal?(zdom(balance), zdom(od_limit))
```

Data invariants are entered into the DICheck template in the form:

```
If  (predicate 1 And
     predicate 2 And
     :
     :
     predicate n)
Then ...
```

In this case, `predicate 1` being `zequal?(zdom(balance), zdom(od_limit))` and `predicate 2` being the Zappa translation of the second predicate in the data invariant of *Account*:

$\forall x : \text{dom } balance \bullet balance \; x \geq od\_limit \; x$

Zappa Function	Z Notation Predicate
`zelement?(x,y)`	$x \in y$
`zequal?(x,y)`	$x = y$
`znot_element?(x,y)`	$x \notin y$
`znot_equal?(x,y)`	$x \neq y$
`zsubset?(x,y)`	$x \subseteq y$
`zpsubset?(x,y)`	$x \subset y$
`zempty?(x)`	$x = \emptyset$
`znot_empty(x)`	$x \neq \emptyset$
`zg?(x,y)`	$x > y$
`zge?(x,y)`	$x \geq y$
`zl?(x,y)`	$x < y$
`zle?(x,y)`	$x \leq y$
`zfor_all_01(d,set,x,rel,y)`	$\forall d : set \bullet x \; rel \; y$
`znot(exp)`	$\neg exp$
`zunion(x,y)`	$x \cup y$
`zintersect(x,y)`	$x \cap y$
`zsubtract(x,y)`	$x \setminus y$
`zcard(x)`	$\# x$
`zequate(x,y)`	$x' = y$
`zmake_empty(x)`	$x' = \emptyset$
`zno_change(x)`	$x' = x$
`zmake_set1(x)`	$\{ x \}$
`zset_comp_01(d,set,x,rel,y)`	$\{ d : set \bullet x \; rel \; y \}$
`zplus(x,y)`	$x + y$
`zminus(x,y)`	$x - y$
`zmult(x,y)`	$x * y$
`zdiv(x,y)`	x div y
`zmod(x,y)`	x mod y
`zneg(x)`	$- x$
`zmake_map(x,y)`	$x \mapsto y$
`zmake_triple(x,y,z)`	(x,y,z)
`zfirst(x)`	first x
`zsecond(x)`	second x
`zdom(x)`	dom x
`zran(x)`	ran x
`zdom_res(x,y)`	$x \lhd y$
`zdom_sub(x,y)`	x ⩤ y
`zoverride(x,y)`	$x \oplus y$
`zran_res(x,y)`	$x \rhd y$
`zrel_image(x,y)`	x ⦇ y ⦈
`zinverse(x)`	x~
`zhead(x)`	head x
`zlast(x)`	last x
`zfront(x)`	front x
`ztail(x)`	tail x
`zsquash(x)`	squash x
`zrev(x)`	rev x
`zconcat(x,y)`	x ⁀ y
`zextract(x,y)`	x ↿ y
`zfilter(x,y)`	x ↾ y

Figure 7.4: Zappa Z Functions.

Which translates in Zappa to:

```
zfor_all_01(x, zdom(balance), balance, >=, od_limit)
```

`zfor_all_01` is one of what will eventually be a library of multi-purpose quantified expression in Zappa, and has the format `zfor_all_01(dummy_variable, set, value, relation, value)`. `zeroZ` is a typed constant, integer 0, supported by Zappa.

The translation of the last data invariant predicate in *Account*:

$\forall\, x : \mathrm{dom}\ od\_limit \bullet od\_limit\ x \leq 0$

is `zfor_all_01(x, zdom(od_limit), od_limit, <=, zeroZ)`

Once the translated data invariant has been entered the user clicks on the Window Close Box of the Function Editor (the small grey box in the top left hand corner of the template window). The user is then prompted to save the function. If the function has been entered correctly, the user clicks on the YES button and the data invariant is entered into the animation, otherwise the user can continue to edit or correct the invariant.

All that is left is to create the dummy variable, *x*, refered to in the quantified expressions of the data invariant. The user presses the `Create Local/Dummy Variable` button in the ANIMATOR screen and enters `x` as the variable name. Pop-up windows appear allowing the user, as when the state variables were created, to select the form, structure and type of the variable, in this case `tuple`, `A` and `ACC_NO`. Zappa then enquires if the user wishes to give the variable a value. In this case, since the variable is not a constant, the user clicks on NO.

The next stage in the animation process is to animate the initial state of the banking system, *InitAccount*, which is given below:

InitAccount

Account

---

balance = ∅
od_limit = ∅

The user presses the Create Initial State button and enters the schema name at the prompt. As for the data invariant, a template is provided for animation. The user follows the same procedure as before to enter the Function Editor and now edit the simple initial sate schema template. This time the predicate to be translated and entered into the template is:

balance = ∅
od_limit = ∅

which once translated into Zappa becomes

```
zmake_empty(balance);
zmake_empty(od_limit);
```

Note the semi-colons, used at the end of a Zappa predicate when not nested within a logical construct such as `If...Then...`

If the user now switches to the ANIMATION screen two new buttons will have appeared: `InitAccount` and `DICheck`. During an animation `InitAccount` can be pressed to initialise the system state. `DICheck` can be used, after an operation is carried out for example, to check that the system data invariant is still intact (i.e. the animation can highlight inconsistancies between post conditions and the data invariant).

At this stage the user might wish to use the Show Variable button (available on the ANIMATION and ANIMATOR screens) to display the state variables. A pop-up menu lists the state variables for one to be selected and its structure and current value displayed in tabular form. At this stage both balance and od_limit should be empty.

The next stage in the animation process is to animate an operation schema. The operation schema to open an account in the banking system is given below:

Open

$\Delta Account$

$new? : ACC\_NO$

$odl? : \mathbb{Z}$

$new? \in \text{dom } balance$

$odl? \geq 0$

$balance' = balance \cup \{ new? \mapsto 0 \}$

$od\_limit' = od\_limit \cup \{ new? \mapsto -odl? \}$

To animate the schema, *Open*, the user firstt has to create the input variables *new?* and *odl?* The user clicks on the `Create Input Variable` button on the ANIMATOR screen and enters the first variable name, `new?`, at the prompt. A pop-up menu then allows the user to select the type of the input variable, in this case `ACC_NO`. Because the user is creating an input variable, Zappa asks for a prompt for the input to be used as part of the user interface for the animation. In this case the user might enter `Enter a New Account Number` as an appropriate prompt for `new?` A similar procedure is followed to create *odl?*

Note that input variables, once created, are available for inclusion in any schema that might be animated, they are reuseable and not the sole property of any one schema.

The user now presses the `Create Operation Schema` button and enters the schema name, `Open`. After answering YES to inputs a multiple selection menu allows the user to highlight the inputs, from the list of input variables so far created, that are required for inclusion in the operation schema. In this case the only input variables created are `new?` and `odl?`, both required by `Open`, so the user highlights both and clicks on OK.

The user then follows the editing procedure for templates, via KTOOLS as before, this time to enter the predicate section of *Open*. (Note that the user can press the KTOOLS button at any time to edit schemas, for example, if syntax errors are reported.) The KAL function template for `Open` (noting that the template is general for operation schemas, excepting the `zinput` lines, generated by Zappa to provide the input variable inclusions particular to `Open`) is given below:

```
{
zinput( GetNthElem( Open:Inputs, 1);
zinput( GetNthElem( Open:Inputs, 2);
If NULL
      Then    {
            zmessage(" ");
            zend( );
            TRUE;
            }
      Else      {
            zend?( );
            FALSE;
            };
}
```

Preconditions are then entered by the user in place of the NULL in the template and separated by the word `And` and postconditions are entered after the curly bracket after the Then. The postconditions are put within brackets, ( ), in much the same way the data invariants were. As before the user translates lines of Z into lines of Zappa. Thus, the first precondition:

$$new? \in \mathrm{dom}\ balance$$

for example, becomes:

```
zelement?(new?, zdom(balance))
```

whilst the postcondition

$$balance' = balance \cup \{ new? \mapsto 0 \}$$

for example, translates into Zappa as:

```
zequate(balance, zunion(balance, zmake_set1(zmake_map(new?,
zeroZ))));
```

After translating and entering the Z predicates the user then gives the operation a message to output, as part of the animation interface. The message is entered between the quote marks in `zmessage(" ");` in the template, in this case a message such as `Account Opened`.

Once the user is happy with the animated schema the small grey button in the top left hand corner of the Function Editor is clicked, followed by clicks on Close and YES in the prompt to save the Function.

The next stage is to animate the error schemas associated with *Open*; *AlreadyExists* (the error being that the account number entered into the operation is already in use) and *NegOdLimit* (the error being that a negative overdraft limit is entered into the operation by mistake). The error schemas are not given here as the process used to create them is much the same as that used to create operation schemas, excepting that the `Create Error Schema` button is used, inputs are not required (they are provided by association with the operation schema) and animated error schemas are associated with `ErrorSchemas` in the OVERVIEW.

Returning to our example, the next step, assuming that the error schemas *AlreadyExists* and *NegOdLimit* have been animated, is for the user to animate the robust *Open* operation specified by *ROpen*:

$ROpen \mathrel{\widehat{=}} Open \lor AlreadyExists \lor NegOdLimit$

The user presses the `Create Robust Op Schema` button and enters `ROpen`. The Function Editor is then invoked in the same way as before, this time to select and edit the robust operation template for `ROpen`. This time schema names are entered into the template in place of a `NULL` statement, remembering to end each schema name with brackets and to separate them with the word `Or`. Thus, in our example, the user edits the template as shown below:

```
If      ( Open()            Or
          AlreadyExists()   Or
          NegOdLimit() )

Then TRUE
```

At this stage the animation can be used, by the developer and client, to investigate the operation to open account. In the ANIMATOR window a button will appear upon the creation of a robust operation schema, in this case a button for `ROpen`. This can now be pressed to execute the animated robust operation. Accounts can be opened, entering appropriate values for account numbers and overdraft limits. The robustness of the operation can be tested (Zappa will inform the user if values of data have been entered that have not been catered for in the specification, by displaying a message telling the user that the operation is not robust) and state variables examined via the `Show Variable` button. In addition, Zappa will also carry out full type checking and syntax checking, halting execution and informing the user if there is a type miss-match or syntax error.

The user now continues to animate the rest of the banking system specification, using the procedures described above, to create variables, animate operation and error schemas, and to combine operation and error schemas to animate robust operations.

7.5 Behind the Scenes.

Zappa introduces the concept of the *sort* of a variable in Z specifications to implement the strict data typing in Z.

A declaration of a variable in Z implies its type, which may either be a basic type or constructed from basic types using type construction operators. Further, variables may be given certain properties by the implicit imposition of constraints on their type. For example, the declaration, $numbers : \mathbb{P}\,\mathbb{N}$, is identical to the declaration, $numbers : \mathbb{P}\,\mathbb{Z}$, given the constraining predicate, $\forall n : \mathbb{Z} \mid n \in numbers \bullet n \geq 0$. Similarly, the declaration of a function, $f : X \nrightarrow Y$, is equivalent to the declaration of a relation, $f : X \leftrightarrow Y$, given the constraining predicate,

$$\forall x : X; y_1, y_2 : Y \bullet (x \mapsto y_1) \in f \wedge (x \mapsto y_2) \in f \Rightarrow y_1 = y_2.$$

Similarly, a sequence in Z is a function with a constraint on the domain of the function.

Thus, although variables in Z may have the same type, we might say that they can be of different *sorts*. Conversely, variables of the same *sort* may be of different type if the basic types used to declare the variable are different.

In Zappa, sort codes are used to identify sorts of variables. For example, an element of a basic type is sort 10, a tuple of type $X \times Y$ is sort 11, a variable of type $\mathbb{F}\,X$ is sort 21, a relation declared using $X \leftrightarrow Y$ is sort 22, a variable declared using $X \nrightarrow Y$ is sort 71 and a sequence of elements from a basic type is sort 41.

Variables in Zappa are represented by Kappa-PC objects of the same name. Kappa-PC objects have slots which can be given slot names and in which information, such as sort codes, can be stored. In Zappa each variable has a slot for its sort code and a slot or slots for the basic type or types used to construct the variable and a slot or slots for its value. For example, a variable declared in Z using $X \nrightarrow Y$ would be represented as an object with the following slots and slot values:

Slot Name	Slot Value
sort	71
Atype	X
Btype	Y
Aelems	list of domain elements
Belems	list of corresponding range elements

In Zappa, KAL functions corresponding to the predicates and set operations of Z's Mathematical Tool-kit use the sort and basic type slots of variables to determine the type correctness of an expression. Retrospectively, within the animated invariant predicate of a system state, Zappa uses sort to flag violations of implicit variable constraints. Finally, sort is used by Zappa to select suitable algorithms to test predicates or carry out set operations.

KAL code for Zappa is given in the volume of appendices.

7.6 Use of Zappa in the Classroom.

Zappa has been used by the author in the teaching of animation to students on the HND Computing Mathematics and MSc Engineering Information Technology courses at Sheffield Hallam University.

The tool has been tested in laboratory workshops where students have been supplied with a simple Z specification and instructions on the use of Zappa with respect to the specification, along with details of the Zappa Z Functions required to animate the specification (a tutorial guide to Zappa, used in student workshops, is included in the volume of appendices). Results were mixed, with some students able to produce a working animation with apparent ease and others, possibly still having difficulties with the Z notation, struggling to make progress.

Zappa, and the investigation of animation using Kappa-PC have been subjects of several student projects supervised by the author, two of which have involved the student directly in the author's research [Dhe93, Jac94].

Zappa has also been used as part of a learning contract by one MSc student to animate GEORGIS, a British Rail track failure database (which was mentioned in chapter 3).

7.7 Strengths and Weaknesses of Zappa.

Zappa is an animator CASE tool, albeit a prototype. It aids the software developer in the production of an animation of a Z specification. The resulting animation can be used by the developer to investigate properties of a specification and, more importantly, the animation can be used to demonstrate the essential features of a specification to a software client or user to assist in the process of validation.

We see that Zappa has four key strengths:

1. A developer's interface, ANIMATOR, to aid in the systematic production of an animation. Many structures in the animation are created automatically by ANIMATOR, thus reducing the workload of the developer.

2. A client's interface, ANIMATION, to aid in the interaction between the client, the developer and the Z specification, for validation and development purposes. As well as providing an interface to system operations and allowing the developer to test operations for robustness, it provides a facility for the developer to test data invariants and a facility to initialise the system state.

3. Lines of Z translate, in a staightforward way, into lines of Zappa. Little or no programming skills or intuitive leaps are necessary as translation and animation is a matter of selecting equivalent Zappa Z functions and following instructions.

4. Zappa implements the srict variable typing of the Z notation and introduces the concept of a variable's *sort* to enforce any implied constraints on Z variables.

Nevertheless, we recognise Zappa's limitations.

Zappa has been used to animate a variety of specifications successfully but some specifications have not been well suited, as written, to be animated by the tool. The Birthday Book specification in Spivey [Spi92], for example, uses the state data invariant to implicitly specify operation postconditions, something that Zappa cannot do. Zappa requires specifications to follow a certain conventional style, although we feel that this constraint is not excessive and that, if necessary, specifications could be rewritten to conform to Zappa's requirements. For example, Zappa requires that all pre and postconditions are explicit. This may give rise to slightly lengthier specifications than otherwise possible but it does have the benefit of encouraging the developer to consider such conditions carefully. Also, by making all pre and postconditions explicit within the specification, proofs concerning the specification can be more easily formulated.

Zappa also requires that specifications are entirely deterministic, but we feel that this is perhaps not surprising given that the specification is to be executed. It might be possible to incorporate non-determinancy into an animation by partial execution, whereby non-determinant outcomes are reported via the animation interface.

The implementation of Z's schema calculus is limited, being restricted to the basic logical operatives available in Kappa-PC, and Zappa can animate only one state schema - complex specifications involving multiple state schemas have to be treated as specifications with one, monolithic, state schema.

Zappa currently implements a limited, but we feel useful, range of Z variable types (see figure 7.3). It is envisaged that this range will be extended and it will always be possible for an experienced Kappa-PC user to implement a type not currently supported. Although there is a fair degree of reuasblity within Zappa's Mathematical Tool-kit library of algorithms (function override, for example, is implemented as per its definition in Spivey [Spi92] using more basic set operations, and needs little further consideration), the addition of a new variable type will involve the creation of Tool-kit algorithms to cater for it.

REFERENCES IN CHAPTER 7

Spi92. Spivey, J. M., *The Z Notation: A Reference Manual*, 2nd Edition, Prentice Hall International, 1992.

Kap90. Kappa-PC Users Guide, InteliCorp Inc., 1990.

Coo92. Cooper, D., Mardell, J., Meehan, A., Norcliffe, A. and Valentine, S., *A Z Readers Course*, Shefield Hallam University Pavic Publications in Assn. with the D.T.I., 1992.

Dhe93. Dhers, B., *Animating Z using Kappa-PC*, MSc Engineering Information Technology Project Report, Andrews, S.J., Project Supervisor, Sheffield Hallam University, 1993.

CHAPTER 8: A CRITICAL REVIEW OF OPERATOR

La distance n'y fait rien;
il n'y a que le premier pas qui coûte.
Marquise du Deffand,
letter to d'Alembert, 1763.

8.1 Introduction.

The research presented in this thesis has been carried out with the specific goal of achieving the three aims set down in chapter 1. They were,

"To investigate the issues involved in creating and demonstrating formal specifications of information systems.",

"To develop a systematic approach to creating formal specifications of such systems." and

"To investigate ways of animating such specifications."

The culmination of the research work has been the development of the OPERATOR approach to creating Z specifications together with the development of the prototype CASE animator, Zappa. These two research outcomes arose in response to the issues that were identified by addressing the first of our aims.

In this chapter we shall evaluate the development of OPERATOR against the first and second of our aims by listing the issues involved in creating formal specifications that have been identified by the research, and to consider how these have been addressed by the development of OPERATOR.

In the next chapter we shall evaluate the development of Zappa against the first and last of our aims in an analogous fashion.

8.2 Evaluation of the Research: Issues involved in creating Z specifications

The issues involved in creating Z specifications are those aired in chapters 3 and 4. Essentially they are the following:

- Although the use of Z to develop a specification, and then refine it to code, is regarded as applying a formal method, the reality is that there is no systematic method being used at all. The emphasis has been on the development of notations rather than methodologies. See McDermid [Der87] and Jackson [Jac87].

- This lack of any method means that acquiring writing skills in Z is more difficult in comparison to the acquisition of reading skills. This fact will have been observed by most teachers of Z. At Sheffield Hallam University, students acquire writing skills by first acquiring reading skills. See, for example, Cooper et al [Coo92].

- All the classic 'abstraction bottleneck' problems emerge when trying to teach would-be software engineers how to use Z to develop system specifications. See Norcliffe [Nor93]. Going straight from requirements to written Z is not easy.

- This lack of a method means that tools to ease students through this abstraction bottleneck, or to assist the developer to capture user requirements simply and effectively, do not as yet exist. See Plat [Pla92]. The building of a tool requires at least some well understood step by step approach.

- In contrast, structural methods, with their well-defined methods, diagrams and tools, do not appear to suffer from these problems (see Chapter 2). Perhaps what is needed is a method based on diagrams for creating Z specifications.

- Communicating with the client at all stages of requirements capture is important if the right system is to be built. The Z notation, whilst being a precise language with which developers can communicate with one another to develop a system correctly actually inhibits communication between developer and customer unless the latter is Z literate. See Jackson [Jac87].

8.3 Evaluation of the OPERATOR approach

If we look at the OPERATOR approach to developing Z specifications, then considering each of the six points above, we see that:

- OPERATOR does represent the beginnings of a method. It has well defined steps associated with it, i.e. O,P,E,R,A,T,O and R steps. These steps can be followed systematically to develop a range of system specifications. The method can be easily taught and learnt and is therefore transferable, as stated in [And95]. Application of the method appears to give a high level of reproducibility in the brief student trial carried out so far.

- The trials with students have suggested that the OPERATOR approach does help students to use Z systematically to develop specifications. It therefore facilitates the acquisition of Z writing skills.

- By being systematic and incremental in its approach the OPERATOR method does appear to ease students, and would-be software engineers, through the abstraction bottleneck. Because the approach is teachable it represents a method which students can follow to build the system state schema. Large intuitive leaps are not needed in using the method and, once the state schema has been obtained, the rest of the specification follows relatively easily.

- Although OPERATOR is a paper-based method, there is no reason why the method should not be embedded in software to create a CASE tool.

- OPERATOR does draw heavily on the ideas of structured methods. However, it is free standing as a method and does not require prior knowledge of any proprietary structured method. This is not to say that there may not be advantages in integrating formal methods with existing structured methods (see Polack [Pol91, Pol92]), and valuable work is being carried out in this area [Sem91, Ran91, Ste90].

- The diagramming notation associated with OPERATOR does appear to help students, although evidence is anecdotal as yet. However, the operation diagrams have all the attributes of data flow diagrams in structured methods, and therefore must posses all their advantages. In contrast we would claim that the entity diagrams - being free of any relationships other than *comprised of* and *has/have* - are intuitively more simple than the entity relationship diagrams of structured methods, and are easier to create and work with. We would claim also that OPERATOR, as a method, is simpler to work with than the methods integration approach. The latter is not a seamless joining of two methods to create a new approach. It capitalises on the strengths of both approaches, but inherits all the problems of integrating two different methods. OPERATOR was developed solely with the aim of helping students to create Z specifications and as a result is not cluttered with any unnecessary paraphernalia.

- The diagrams of the OPERATOR method, whilst helping the specifier to think in concrete terms about the system, have the additional advantage of being simple and sufficiently intuitive to be used in communicating with the client. All the advantages that diagrams bring to structured methods apply equally to the OPERATOR approach.

OPERATOR may be criticised in that it tends to formulate a data structure in terms of functions and relations rather than sets with appropriate invariants, and this might be seen as restricting the choices of a software developer. For example in the case of a simple security system where identification codes of valid members of staff are stored in such a way as to indicate the whereabouts of staff (see Chapter 4), OPERATOR produces a function relating identification codes to whereabouts:

$staff\_whereabouts : STAFF\_ID \rightarrowtail\!\!\!\!\!\!+ \; WHEREABOUTS$

where *WHEREABOUTS* is an enumerated type,

$WHEREABOUTS ::= staff\_in \mid staff\_out$

The system could just as easily be modelled by sets of identification codes, each set representing a different whereabouts:

$staff\_in, staff\_out : \mathbb{P}\ STAFF\_ID$

with the constraint

$staff\_in \cap staff\_out = \varnothing$

It would seem that a general point can be made; functions that have ranges that can easily be enumerated can be modelled, alternatively, by sets.

However, we might point out that, at some stage in the implementation of such a system, refinement of these sets may well produce a more functional view (see Wordsworth [Wor92], for example). We would also note that, although OPERATOR naturally produces specifications that are amenable to our animator, the animator copes just as easily with a set based approach as it does with a function or relation based approach.

In summary, the success of a software development project depends heavily on the process of capturing the real requirements of a system and, in large systems, the appropriate partitioning of the system. As we have already stated, we feel that OPERATOR with its graphical front end, and its system partitioning capability has potentially much to offer the Z user. It has evolved from a limited abstract method to the fuller method it is today. Much remains to be done developing the method further and specific ideas for further research are presented in Chapter 10.

REFERENCES IN CHAPTER 8

Der87. McDermid, J., *The Role of Formal Methods in Software Development*, Journal of Information Technology, Vol. 2, No. 3, Sept. 1987.

Jac87. Jackson, M., *Power and Limitations of Formal Methods for Software Fabrication*, Journal of Information Technology, Vol. 2, No. 2, June 1987.

Coo92. Cooper, D., Mardell, J., Meehan, A., Norcliffe, A. and Valentine, S., *A Z Readers Course*, Sheffield Hallam University Pavic Publications in Assn. with the D.T.I., 1992.

Pla92. Plat, N., van Katwijk, J. and Toetenel H., *Application and Benefits of Formal Methods in Software Development*, Software Engineering Journal, Sept. 1992.

And95. Andrews, S., and Norcliffe, A., *A Systematic Approach to Writing Simple Z Specifications*, Proceedings of The 3rd Annual Conference on the Teaching of Computing, Dublin, August, 1995.

Pol91. Polack, F., Whiston, M. and Hitchcock, P., *Structured Analysis - A Draft Method for writing Z Specifications*, Proceedings of the sixth annual Z User Meeting, Dec. 1991.

Pol92. Polack, F., *Integrating formal notations and systems analysis: using entity relationship diagrams*, Software Engineering Journal, Sept. 1992.

Sem91. Semmens, L. and Allen, P., *Using Yourdon and Z: an approach to formal specification*, Proceedings of the fifth annual Z User Meeting, Nicholls, J.E., Ed., Springer-Verlag, London, 1991.

Ran91. Randell, G., *Data Flow Diagrams and Z*, Proceedings of the fith annual Z User Meeting, Nicholls, J.E., Ed., Springer-Verlag, London, 1991.

Ste90. Stepney, S., *Specifying entity relationship diagrams in Z*, ORCA/Logical, 1990.

Wor92. Wordsworth, J. B., *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering*, Addison-Wesley, 1992.

CHAPTER 9: A CRITICAL REVIEW OF ZAPPA

Give us the tools,
and we will finish the job.
Sir Winston Churchill, 1941.

9.1 Introduction.

As we have previously stated, the research presented in this thesis has been carried out with the specific goal of achieving the three aims set down in Chapter 1. In this penultimate chapter we look at what has been achieved with respect to the first and third of these aims.

In the next section we list the issues that have been identified in the thesis that relate to demonstrating formal specifications of information systems. We then expand on observations made in Chapters 6 and 7, and consider the extent to which the prototype CASE animator Zappa, and the animation work carried out with Crystal, has addressed these issues. We then compare the approach that we have taken, particularly in the development of Zappa, with the approach and views of others in the field. Finally, we look critically at the style of Z that is required before an animation in Zappa is possible.

9.2 Evaluation of the Research: Issues involved in demonstrating Z specifications.

The issues involved in demonstrating Z specifications are those aired in Chapters 3, 5, 6 and 7. The issues can be listed as follows:

- The importance of validating a specification against user requirements is vital. Formal methods, in principle, enable a system to be built correctly. They do not guarantee that the correct system will be built. Checking that the specification has captured the user requirements is essential.

- Checking, therefore, that the mathematical model of the system is capturing the correct user requirements is a vital part of developing a system using formal methods. Animation may be one means of satisfying the user and the developer that what is specified is valid.

- Exercising the Z specification by dynamic testing via a rapid prototype or animation has many advantages. Work carried out reasoning about a static specification may not clearly reveal all aspects of system behaviour.

- Because Z will not execute, as yet, a translation to some form of executable code is needed. It is essential that this translation is faithful to the Z it represents and easily effected.

- With respect to animators used to demonstrate formal specifications, it is essential that they are effective as validation tools. It is no use having an animator that only helps the developer. It must be able to demonstrate the system to the customer in a way that facilitates the validation of the specification against user requirements.

- Any animator should be easy to use by the specifier.

- An animator should be able to animate a range of specifications. A clear issue is what are the restrictions on specification, if they are to be animated. Clearly the style of the written Z is a factor determining whether a specification can be animated or not, see chapter 7, section 3.

9.3 Evaluation of Zappa and the Crystal approach.

This section evaluates the Zappa and Crystal approach in the light of the issues raised in the previous section. A further evaluation, comparing the approach to that taken by others is given in the next section.

Both the Crystal approach and Zappa provide dynamic demonstrations that may be used to show the essential features of systems specified in Z.

Zappa, with its interfaces, automation of animation and library of Z functions certainly has the beginnings of a tool that can aid the developer in demonstrating the system for the purposes of requirements validation.

Both Zappa and the Crystal approach attempt to demonstrate a mathematical model of a system by remaining faithful to the Z notation, although it is not easy to say how faithfully the Z notation has been implemented. Types are not addressed using the Crystal approach, for example, and variables have global scope in Zappa.

The efficacy of Crystal and Zappa animations has not been evaluated in depth in terms of validation of a client's system requirements, although the author has had animations demonstrated to him by students, notably an intelligent multi-storey car park system (Crystal) and a rail failure data-base, GEORGIS (Zappa). Both animations provided a clear understanding of the systems being developed.

Both the Crystal approach and Zappa have proved themselves fairly easy to use, with students able to produce sensible animation code in tutorials and working animations in workshops, with little or no prior knowledge.

The issue of the style of Z required for animation using Crystal or by Zappa is one for debate. Restrictions are certainly there, but the impact they may have on formal software development, negative or positive, is, at this stage, difficult to resolve. We shall discuss the issue of style in a later section.

9.4 Comparison of the Zappa approach with related work.

From the points listed in 9.2, and the nature of the Zappa tool that has been developed, it is clear that the view taken of animation in this thesis has been one of providing a rapid prototype of the system, that is faithful to the original Z specification, that can be used to demonstrate the functionality of the system to a would-be customer or client. There are different views to animation and in this section we look at the work of others on animation and contrast their work with the work presented here.

To help illustrate a different view of animation let us consider the specification of two functions, a square root function and a square function. Their specifications are given below.

$$\begin{array}{|l}
sqrt : \mathbb{N} \nrightarrow \mathbb{N} \\
\hline
\forall x, y : \mathbb{N} \bullet y = sqrt(x) \Leftrightarrow y * y = x
\end{array}$$

$$\begin{array}{|l}
square : \mathbb{N} \rightarrow \mathbb{N} \\
\hline
\forall x : \mathbb{N} \bullet square(x) = x * x
\end{array}$$

Mathematics can be very abstract for the purpose of capturing system requirements, allowing the developer to concentrate on the *what* for ease of capture, but the *how* can often be more useful. In the function definitions above the specification of the square root function is a declarative one that is too abstract as it stands to be of much use for the immediate purposes of implementation. The definition of the square function, however, can be implemented as mathematically specified. When considering demonstration, the specification of the square root function may cause problems. It is not executable, as such, but could be considered 'animatable' in terms of verification. Indeed, this view of animation is described by West and Eaglestone [Wes92] who have used Prolog to create 'simulations' of Z specifications by capturing their mathematical structure. They identify characteristics of Z such as schema signatures and predicates, data types and variable decorations (a similar approach to our own), and describe how these can be simulated in Prolog. The resulting translations are of a non-deterministic nature and provide a way of checking what is expected to be true of a specification, given a state space and a predicate. This compares with traditional software testing procedures where chosen input and output pairs are verified by execution.

It is worth noting, as it relates to the issues concerning the executability of mathematics (see Hayes and Jones, *Specifications are not (necessarily) executable* [Hay89]), that West and Eaglestone concluded that there was no straightforward mapping between Z and Prolog - a mapping had to be manufactured using some characteristics of Z:

> *"... the simulation depends on the characteristics of a specification being within the bounds of* [their] *stated rules ... A Z specification gives a logical relationship, whereas Prolog, although in theory a declarative language, in practice does rely on the textual sequence of the code. The lack of data types also means that Prolog sets have to be implemented by lists. These factors could limit the subset of Z that is capable of translation by this technique and its possible mechanisation."*

We had similar problems with ZAL (the programming language of Kappa PC) in identifying a subset of Z that is translatable and we have been unable to be explicit in terms of what can and what cannot be demonstrated through animation (see suggestions for further work in the final chapter). We also note that sets are typically implemented as lists whichever programming language is chosen (Morrey, Siddiqi, Buckberry and Hibberd [Mor92] use the same method with the functional programming language LISP). This in itself is a concrete refinement of an abstract object, imposing an order on elements in a set. The issue of whether Z can wholly and satisfactorily be animated is clearly a fundamental one. A final point on this area of executability is provided by Valentine [Val94] when he discusses the rationale behind Z--, his executable subset of Z:

> *"Many studies of refinement and abstraction use different notations for the specification and the program. This obscures the fact that a program is a special case of a specification, and creates pointless extra work in the translation. Some work has been done on creating specification subsets of existing programming languages. In general this may produce rather messy results. Z-- has been developed as the programming language which is a subset of Z."*

It is clear that there is more than one view or purpose of animation. West and Eaglestone, above, have developed animation as a means of specification verification. Prolog, with its capability of backward and forward chaining lends itself to this approach and that of providing animation in terms of enumerating all possible solutions to a logical problem - give a state space and a constraining predicate an animation can be a means of providing instances within the state space that satisfy the predicate.

Our approach was to develop animation as a means of demonstrating Z specifications to the non-mathematically literate, a possible means of validating perceived with actual system requirements. Essentially, this is also the approach of

Morrey et al. [Mor92] although they also see animation as a direct support for the development of a Z specification:

> *"Not only does the package* [their LISP based Z animator] *provide facilities for users to validate their own specifications, but its interactive technique also encourages and supports an experimental approach as a valid technique for the development of Z specifications."*

Cooling and Hughes [Coo94] have also identified the problem of communicating formal specifications to non-specialists, but have taken a more traditional view of animation, in that they aim to show the meaning of such specifications in computer generated pictures. However, as one might expect, their work is mainly concerned with the animation of real time systems (our problem domain is information systems) and hence further comparison with Zappa is not easily made.

Mukherjee [Muk95] also adopts a validation approach to the animation of VDM, with the view that a specification should be demonstated to the user. However he points out that there is a danger in that the effort involved in animation may be as much as the effort involved in refining the specification into a program, unless the specification is written in a procedural style.

Johnson and Saunders [Joh89] view animations as stepping stones towards final implementation. They show how Z can be translated into a 'specification-like' functional program, producing a prototype that forms the basis for further refinement. They see the main advantage of this approach being that the engineer is working in an executable design language, where the engineer can formally refine the animation rather than going back to the original specification.

Sherrell and Carver [She93] again take a validation approach through rapid prototyping, implementing Z directly in the functional programming langauge Haskell. Worthy of note is the fact that they animate a similar style of Z to that which Zappa best animates, namely one that is deterministic and procedural in appearance.

It may be worth noting, given that Zappa views animations as procedural, deterministic demonstrations, that Zappa could be used to test non-deterministic or implicit specifications if the animation is developed in a particular way. Taking the example of the square root function above, if the quantified variables, *x* and *y*, were animated as state variables and the axiom was animated as a data invariant, values can easily be assigned to *x* and *y* and tested using Zappa's data invariant checker.

One area of commonalty in the various approaches to animations, including our own, appears to be the use of a library of Z operations; essentially the complete or partial implementation of Spivey's mathematical toolkit [Spi92]. Knott, for example, uses a library of Prolog procedures [Kno90, Kno92] which is now well developed and may be considered a benchmark for future work. He takes the theory of animation to encompass wider uses, say with working mathematicians and people interested in other executable mathematical notations. It could be argued that Zappa offers the possibility of investigating properties of discrete mathematics by animating sets, relations, functions and the like, although some consideration has to be given to the fact that Zappa is an environment for animation of software specifications - the developer's interface uses terminology and procedures commensurate with that activity. A cut down version of Zappa to cater for Computer Aided Learning of discrete mathematics, for example, could be produced with some modification to the existing tool, with new uses being made of the animation interface tailored to specific topics or lessons.

Knott also shares the goal of using a style of programming that relates closely to the mathematics, but goes further in that ideas of proof and correctness preserving transformation should be catered for. It is not clear that this level of formality could be achieved with adaptations to Zappa.

9.5 The style of Z required by Zappa.

The style of Z that is required by Zappa may be considered restrictive - ZAL does not allow flexibility in terms of such things as currying functions and 'lazy evaluation', things that Knott sees as advantages of using a functional programming language or the backtracking provided by Prolog to materialise alternative solutions. Zappa requires a Z operation schema to be deterministic in nature and procedural in appearance. Preconditions must be evaluated before postconditions are implemented. Given particular inputs to an operation, there can be either no change of state or one change of state. This may, to some extent, be because Zappa has been developed to animate software specifications of information systems, not for the execution of mathematics in general. It is also important to Zappa that post conditions are evaluated strictly from left to right, with a single variable to be evaluated. For example,

$x \cup y = z'$

becomes

$z' = x \cup y$

A statement such as

$z' \cup y = x$

would, as it stands, be evaluated as a true or false statement and could not change the state of the system. It could be rewritten as

$z' = x \setminus y$

if that was the intention in the first version. Zappa either tests for logical equality *or* assigns a new value to a variable; is *x* equal to *y*? *or* make *x* equal to *y*.

A statement such as

$x' \cup y' = z$

is more problematical. A possible deterministic solution could be given by

$x' = z \setminus y \wedge y' = y$

Statements that do not involve equality can only be considered by Zappa as logical tests. For example

$x' \subset z$

would be a test on an after state variable and would not determine a value for x'.

Further, the notation used must be explicit. Take the statements

$staff = staff\_in \cup staff\_out$
$staff\_in' = staff\_in \cup \{ staff? \}$

The first is a data invariant of a simple security system, the second a post condition for a particular system operation. The conventions of schema inclusion in Z may give rise to the hidden post condition

$staff' = staff\_in' \cup staff\_out'$

in the system operation. Zappa does not follow that convention in that data invariants are used periodically by the user or developer to test the system state. For reasons of speed of execution, the data invariant is not automatically tested before and after an operation (although this is easily possible). The hidden post condition could be included by including it as an assignment of *staff'* after the statement assigning *staff_in'* but the preferred approach would to be explicit within the operation schema, i.e. to mention *all* after state variables. Hence

$$staff' = staff \cup \{ staff? \}$$

would also appear in the operation schema predicate.

REFERENCES IN CHAPTER 9

Wes92. West, M., M. and Eaglestone, B., M., *Software development: two approaches to animation of Z specifications using Prolog*, Software Engineering Journal, July, 1992.

Hay89. Hayes, I., J. and Jones, C., B., *Specifications are not (Necessarily) Executable*, Software Engineering Journal, Nov., 1989.

Mor92. Morrey, I., Siddiqi, J., Buckberry, G. and Hibberd, R., *A Systematic Approach and a Toolset to Support the Construction and Animation of Formal Specifications*, Internal Document, Sheffield Hallam University, 1994.

Val94. Valentine, S., *The Z-- Language*, Internal Document, University of Brighton, March, 1994.

Coo94. Cooling, J., E. and Hughes, T., S., *Making formal specifications accesible through the use of animation prototyping*, Microprocessors and Microsystems, Vol. 18, No. 7, Butterworth-Heineman Ltd., September, 1994.

Muk95. Mukherjee, P., *Computer-aided validation of formal specifications, Software Engineering Journal, July*, 1995.

Joh90. Johnson, M. and Sanders, P., *From Z Specifications To Functional Implementations*, in Nicholls, J., E., (Ed), Z User Workshop, Oxford 1989, *Workshops in Computing*, Springer-Verlag, 1990.

She93. Sherrell, L., B. and Carver, D., L., *Z Meets Haskell: A Case Study*, IEEE Software, Vol. 10., 1993.

Spi92. Spivey, J. M., *The Z Notation: A Reference Manual*, 2nd Edition, Prentice Hall International, 1992.

Kno90. Knott, R., D., Krause, P., J. and Byers, P. J., *Animating Set-Theoretic Specifications using Prolog (Collected Papers)*, Computing Sciences Report CS-90-02, University of Surrey, 1990.

Kno92. Knott, R., D., *Using Prolog to Animate Mathematics*, in *Logic Programming: New Frontiers*, Brough, D., R., (Ed), Intellect Books, 1992.

CHAPTER 10: FUTURE WORK

Thou whoreson zed!
Thou unnecessary letter!
William Shakespeare, King Lear, II: 2.

10.1 Introduction.

In this final chapter we list areas of research that will need to be considered to take forward the work that has already been started on writing and animating Z specifications.

For simplicity we look at writing and animation separately. In particular we concentrate on how OPERATOR might be developed further and the research that is needed to improve Zappa.

10.2 Areas for future research - Developing OPERATOR further.

The OPERATOR approach certainly contains the beginnings of a method. However, this method needs to be defined more sharply and the separate steps of the method expanded.

A tighter definition of an object is needed so that system objects can be readily identified at the beginning of the method.

The *has/have* properties of objects that then flow also need tighter definition - and possibly need expanding in scope - so that all system entities can be identified with confidence. Whilst the present method has enabled complete specifications to be drawn up for the relatively simple systems to which it has been applied, there is no guarantee that the method will be capable of identifying all the entities in more complex systems.

The R step of the method is designed primarily to identify binary relations. Again, it is not clear that all systems can be specified in terms of sets of entities and binary relations. An approach for identifying general system relationships is needed as well.

Data modelling is also not well developed in the method as it stands and further research to enable this first vital link with the Z notation is needed.

In essence then, the steps of the method need to be defined yet again and developed further to enable the approach to have wider applicability. We are aware that research on all these aspects of developing OPERATOR further is being carried out by Tamarin Othman at Sheffield Hallam University [Oth95]. Recent discussions suggest that Object Oriented Analysis is proving useful in attacking many of the areas mentioned - particularly when systems are complex.

There is no reason why the OPERATOR method should cease with the identification of state variables and the signatures of operation schemas. Systematic ways of arriving at data variants together with pre and post conditions for specifying operations must now be researched.

The whole issue of addressing complexity and structuring large specifications needs to be researched more fully and clear ways for developing specifications of complex systems formulated. The approach outlined works for the systems considered so far, but needs to be extended to include the concept of different levels of subsystems.

Along with all of this needs to be developed the associated diagramming notation.

Finally, once the OPERATOR method has been more sharply defined and the separate steps expanded, then tool support to enable the developer to use the method easily and consistently needs to be provided. Once again, we are aware that work aimed at supporting the method with a prototype CASE tool is being carried out by Tamarin Othman at Sheffield Hallam University. The outcomes of this research are awaited.

10.3 Areas for further research - Developing the Crystal Approach and Zappa further.

Areas of further development for the Crystal approach and for Zappa have already been briefly indicated in chapters 6 and 7, respectively. A library of Z functions might be supplied for Crystal, written in C and interfaced with the Crystal shell.

Zappa can be expanded by adding algorithms to cater for additional constructed types. The reusability of existing ZAL code in Zappa makes this less of a task as it may at first seem.

At present Zappa assumes that a single state schema is at the heart of a specification and that state variables have global scope. This is inadequate if Zappa is to be able to animate large systems. Provision will have to be made to implement the schema calculus of Z faithfully, schema inclusion in particular.

As we have presented only anecdotal evidence here, work still has to be done to investigate the efficacy of the Crystal approach and Zappa in terms of validation of user requirements.

The restrictions on the style of Z required for animation by Zappa highlights an important and general issue regarding the use and development of animators and other

CASE tools for Z: What in Z is 'animatable' and how restrictions on the writing of Z specifications, imposed by tools, affect the formal software development process are areas that we feel need careful consideration. We are aware that research considering 'animatability', and the definition and specification of an 'ideal' animator for Z, is being carried out by Alistair Jack at Sheffield Hallam University [Jac95].

Formal methods will surely play a part in the future development of software engineering, either as fully developed methods in their own right or as tools within existing structured software engineering approaches. Software systems will continue to become more complex, widespread and safety critical. Formal methods will be required to exert scientific rigour on the software development process, and, with well developed techniques and computer-based tools to aid in the process, will become ever more important.

Finally, we hope that what has been presented here will add a little to the experience and knowledge of practitioners and researchers in this challenging field.

REFERENCES IN CHAPTER 10

Oth95. Othman, M., T., A., B., registration for degree of MPhil: *Developing and validating a methodology and supporting tool for writing formal Z specifications*, Sheffield Hallam University, October, 1995.

Jac95. Jack, A., registration for degree of MPhil\PhD: *Definition and Specification of a Z Animator*, Sheffield Hallam University, July, 1995.

REFERENCES IN ALPHABETICAL ORDER

Alv87. *Alvey Programme Annual Report*, Alvey Directorate, IEE Publishing, 1987.

And90. Andrews, S., J. and Norcliffe, A., *A CASE Tool for Demonstrating Z Specifications*, IEE Colloquium Digest No. 1990/058, April, 1990.

And91. Andrews, S., J. and Norcliffe, A., *An Expert System CASE Tool for Simulating Z Specifications*, Polymodel 13 Conference Proceedings, 1991.

And92. Andrews, S., J., *A Lift System*, a case study in *A Z Readers Course*, Sheffield Hallam University Pavic Publications in Assn. with the D.T.I., 1992.

And93a. Andrews, S., J., *Learning Contracts*, a case study in *Innovations in Mathematics Teaching*, Staff and Educational Development Association Publications, 1993.

And93b. Andrews, S., Edwards, P., Faulkner, B., Hodgkin, L., Norcliffe, A., Smith, F., and Steere, P., *Student Feedback as an Element in Assuring Course Quality*, Proceedings of the 1993 Undergraduate Mathematics Teaching Conference, Shell Centre for Mathematical Education, 1993.

And95. Andrews, S., and Norcliffe, A., *A Systematic Approach to Writing Simple Z Specifications*, Proceedings of The 3rd Annual Conference on the Teaching of Computing, Dublin, August, 1995.

Ash92. Ashoo, K., *The Genesis Z Tool - An Overview*, FACS Facts, The Newsletter of the BCS Formal Aspects of Computing Science SIG, May, 1992.

Bar91. Barden, R., Stepney, S. and Cooper, D., *Report of a Survey into the use of Z*, ZIP document , Logica Ltd., Dec. 1991.

Bau72. Bauer, F.L., *Software Engineering*, Amsterdam: North Holland, 1972.

BBC93. BBC Radio 4, *File on Four* on *Software Engineering*, Oct. 19th, 1993.

Bel92. Bell, D., Morrey, I. and Pugh, J., *Software Engineering: A Programming Approach*, 2nd Edition, Prentice Hall International (UK) Ltd., 1992.

Blu92. Blum, B., I., *Rapid Prototyping of Information Management Systems*, ACM-SIGSOFT Software Engineering Notes, Vol. 7, No. 5, Dec., 1992.

Bra90. Braithwaite, K., S., *Applications Development Using CASE Tools*, Academic Press Incorporated, 1990.

Bro82. Brooke, F., P., *The Mythical Man Month*, Addison-Wesley,1982.

Bur83. Burnham, D., *The Rise of the Computer State*, Redwood Burn, 1983.

Bur87. Burgess, R., S., *Structured Program Design using JSP*, Hutchinson, 1987.

Bux87. Buxton, J. and Marco, A., *The Craft of Software Engineering*, Addison-Wesley, 1987.

CCT90. Central Computer and Telecommunications Agency, *SSADM Directory of Services*, London: CCTA/BCS, 1990.

Coo92. Cooper, D., Mardell, J., Meehan, A., Norcliffe, A. and Valentine, S., *A Z Readers Course*, Sheffield Hallam University Pavic Publications in Assn. with the D.T.I., 1992.

Coo94. Cooling, J., E. and Hughes, T., S., *Making formal specifications accesible through the use of animation prototyping*, Microprocessors and Microsystems, Vol. 18, No. 7, Butterworth-Heineman Ltd., September, 1994.

Cry87. *Crystal: The Expert System Builder*, Users Guide, Intelligent Environments Ltd., Richmond, London, 1987.

Del90. Delisle, N. and Garlan, D., *A formal specification of an oscilloscope*, IEEE Software, Vol. 7, No. 5, 1990.

Der87. McDermid, J., *The Role of Formal Methods in Software Development*, Journal of Information Technology, Vol. 2, No. 3, Sept. 1987.

Dhe93. Dhers, B., *Animating Z using Kappa-PC*, MSc Engineering Information Technology Project Report, Andrews, S.J., Project Supervisor, Sheffield Hallam University, 1993.

Dic90. Dick, A., Krause, P. and Cozens, J., *Computer aided transformation of Z into Prolog*, in *Z User Workshop*, J.E.Nicholls, editor, Workshops in Computing, Springer-Verlag., 1990.

Dil90. Diller, A., *Z: An Introduction to Formal Methods*, John Wiley and Sons,1990.

Dji81. Djikstra, E. W., *Forewood* to Gries, D., *The Science of Programming*, Springer-Verlag, 1981.

Dow92. Downs, E., Clare, P. and Coe, I., *Structured Systems Analysis and Design Method: Application and Context*, 2nd Ed., Prentice Hall, 1992.

Dun94. Dunn, J., *Give them what they asked for*, lead article in *The Production Engineer*, March 24th, 1994.

Ear86. Earl, A. N., Whittington, R.P., Hitchcock, P. and Hall, A., *Specifying a semantic model for use in an integrated project support environment*, in *Software Engineering Environments* (Sommerville, I., Ed.), Peter Peregrinus Ltd., 1986.

Fai90. Fairburn, D., *The Opportunity of the Decade*, paper in *CASE on Trial*, Edited by Spurr and Layzell, John Wiley and Sons Ltd., 1990.

Fis88. Fisher, A.S., *CASE: Using Software Development Tools*, John Wiley and Sons, 1988.

FOR87. *FORSITE Evaluation System*, User's Guide, Alvey FORSITE Project, Sept. 1987.

For90. *Formaliser User Guide (Z Specification Release)*, Version 6.0, Logica Cambridge Ltd., Nov., 1990.

For94. Information and Technology Foresight Panel, *The Delphi Questionaire (Phase I)*, UK Technology Foresight Programme, Office of Science and Technology, Aug. 1994

Gib88. Gibbins, P. F., *What are Formal Methods*, Journal of Information and Software Technology, Vol. 30, No. 3, April 1988.

Gog88. Goguen, J. A. and Winkler, T., *Introducing OBJ 3*, SRI International, 1988.

Gra91. Gravell, A., M., *What is a good formal specification?*, in proceedings of the Fifth Annual Z User Meeting, Nicholls, J.E., Ed., Springer-Verlag, London, 1991.

Ham93. Hammer, M.J., *SSADM Version 4: Project Manager's Handbook*, McGraw-Hill, 1993.

Hay87. Hayes, I., (ed.), *Specification Case Studies*, Prentice Hall International,1987.

Hay89. Hayes, I., J. and Jones, C., B., *Specifications are not (Necessarily) Executable*, Software Engineering Journal, Nov., 1989.

Hoa75. Hoare, T., quote in lectures notes of Norcliffe, A., Sheffield Hallam University, quote dated 1975.

Hoa81. Hoare, C.A.R., *The Emperors Old Clothes*, Comms. of the Association of Computing Machinery, reproduced in *Writings of the Revolution: Selected Readings on Software Engineering*, Yourdon, E., (Ed.), 1982, Yourdon Press, N.Y., 1981.

Hoa85. Hoare, C. A. R., *Communicating Sequential Processes*, Prentice Hall, 1985.

Hoa86. Hoare, T., *Maths adds safety to computer programs*, New Scientist, 18th Sept., 1986.

Inc88. Ince, D., *Z and system specification*, Journal of Information and Software Technology, Vol. 30, No. 3, April, 1988.

Jac85. Jackson, M., A., *Principles of Program Design*, Academic Press, 1985.

Jac87. Jackson, M., *Power and Limitations of Formal Methods for Software Fabrication*, Journal of Information Technology, Vol. 2, No. 2, June 1987.

Jac95. Jack, A., registration for degree of MPhil\PhD: *Definition and Specification of a Z Animator*, Sheffield Hallam University, July, 1995.

Joh90. Johnson, M. and Sanders, P., *From Z Specifications To Functional Implementations*, in Nicholls, J., E., (Ed), Z User Workshop, Oxford 1989, *Workshops in Computing*, Springer-Verlag, 1990.

Jon90. Jones, C. B., *Systematic Software Development using VDM*, 2nd Edition, Prentice Hall International, 1990.

Jor91. Jordan, D., McDermid, J., A. and Toyn, I., *CADiZ - Computer Aided Design in Z*, proceedings of the Fifth Annual Z User Meeting, Oxford 1990, Springer-Verlag, 1991.

Kap90. Kappa-PC Users Guide, InteliCorp Inc., 1990.

Kin89. King, S., *Z and the Refinement Calculus*, Procs. of VDM90 Symposium, Sept. 1989.

Kno90. Knott, R., D., Krause, P., J. and Byers, P. J., *Animating Set-Theoretic Specifications using Prolog (Collected Papers)*, Computing Sciences Report CS-90-02, University of Surrey, 1990.

Kno91. Knott, R., D., *Making Discrete Mathematics Executable on a Computer*, in *The Mathematical Revolution Inspired by Computing*, Johnson, J.H. and Loomes, M.J (Eds.), The Institute of Mathematics and its Applications Conference Series, New Series No. 30, Clarendon Press, Oxford, 1991.

Kno92. Knott, R., D., *Using Prolog to Animate Mathematics*, in *Logic Programming: New Frontiers*, Brough, D., R., (Ed), Intellect Books, 1992.

Lam86. Lamport, L., *LaTeX: A Document Preparation System*, Addison-Wesley, 1986.

Lit92. Litteck, H. J. and Wallis, P. J. L., *Refinement methods and refinement calculi*, Software Engineering Journal, Vol. 7, No. 3, May 1992.

Lov93. Love, M., *Animating Z specifications in SQL*Forms3.0*, in *Z User Workshop, London 1992*, Bowen, J.P. and Nicholls, J.E., Eds., Workshops in Computing, Springer-Verlag, 1993.

Mai87. Maibaum, T. and Sadler, M., *Formal Methods: A Commentary*, Journal of Information Technolgy, Vol. 2, No. 2, June 1987.

Mil87. Mills, H. D., Dyer, M. and Linger, R., *Cleanroom Software Engineering*, IEEE Software, Vol. 4, No. 5, 1987.

Mor84. Morgan, C. and Sufrin, B., *Specification of the UNIX filing system*, IEEE Trans. Software Engineering, Vol. 10, No. 2, 1984.

Mor90. Morrey, I., Siddiqi, J., and Shaw, *Rapid Prototyping of Formal Specifications*, Sheffield Hallam University, to be published, 1990.

Mor92. Morrey, I., Siddiqi, J., Buckberry, G. and Hibberd, R., *A Systematic Approach and a Toolset to Support the Construction and Animation of Formal Specifications*, Internal Document, Sheffield Hallam University, 1994.

Muk95. Mukherjee, P., *Computer-aided validation of formal specifications, Software Engineering Journal, July*, 1995.

Nei91. Neilson, D. and Prasad, D., *zedB: A proof tool for Z built on B*, proceedings of the Sixth Annual Z User Meeting, University of York, 1991.

Nic91. Nicholls, J.E., *A Survey of Z Courses in the UK*, in Proceedings of the *1990 Z User Workshop*, Springer-Verlag, 1991.

Nor87. Norris, M. T., Newsman, P. J. and James P., *A Step-by-Step Guide to Using Formal Methods*, British Telecommunications Enginering, Vol. 5, Jan. 1987.

Nor91. Norcliffe, A. and Slater, G., *Mathematics of Software Construction*, Ellis Horwood Ltd., 1991.

Nor93. Norcliffe, A., *Computer Aided Modelling*, in Proceedings of the Eighteenth Undergraduate Mathematics Teaching Conference, Yardley, P. (Ed.), Shell Centre Publications, 1993.

Oth95. Othman, M., T., A., B., registration for degree of MPhil: *Developing and validating a methodology and supporting tool for writing formal Z specifications*, Sheffield Hallam University, October, 1995.

Par91. Parkinson, J., *Making CASE Work*, NCC Blackwell Ltd., 1991.

Pla92. Plat, N., van Katwijk, J. and Toetenel H., *Application and Benefits of Formal Methods in Software Development*, Software Engineering Journal, Sept. 1992.

Pol91. Polack, F., Whiston, M. and Hitchcock, P., *Structured Analysis - A Draft Method for writing Z Specifications*, Proceedings of the sixth annual Z User Meeting, Dec. 1991.

Pol92. Polack, F., *Integrating formal notations and systems analysis: using entity relationship diagrams*, Software Engineering Journal, Sept. 1992.

Pol93. Polack, F., Whiston, M., and Mander, K., The SAZ Project: Integrating SSADM and Z, in FME '93 - Industrial Strength Formal Methods, Lecture Notes in Computer Science, Woodcock, J.C.P and Larson, P.G., Eds., Springer-Verlag, London, 1993.

Pot91. Potter, B., Sinclair, J. and Till, D., *An Introduction to Formal Specification and Z*, Prentice Hall International (UK) Ltd., 1991.

Pro86. *Turbo Prolog, the Natural Language of Artificial Intelligence*, Borland International Inc., 1986.

Ran91. Randell, G., *Data Flow Diagrams and Z*, Proceedings of the fith annual Z User Meeting, Nicholls, J.E., Ed., Springer-Verlag, London, 1991.

Sem91. Semmens, L. and Allen, P., *Using Yourdon and Z: an approach to formal specification*, Proceedings of the fifth annual Z User Meeting, Nicholls, J.E., Ed., Springer-Verlag, London, 1991.

She93. Sherrell, L., B. and Carver, D., L., *Z Meets Haskell: A Case Study*, IEEE Software, Vol. 10., 1993.

Sla87. Slater, S. (Ed.), *Essential Mathematics for Software Engineers*, Peter Peregrinus Ltd., 1987.

Som92. Sommerfield, I., *Software Engineering*, 4th Edition, Addison-Wesley, 1992.

Spi88. Spivey, J. M., *Understanding Z: A Specification language and its formal semantics*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1990.

Spi88a. Spivey, J., M., *The fUZZ Manual*, J.M.Spivey Computing Science Consultancy, Oxford, 1988.

Spi90. Spivey, J. M., *Specifying a real-time kernel*, IEEE Software, Vol. 7, No. 5, 1990.

Spi92. Spivey, J. M., *The Z Notation: A Reference Manual*, 2nd Edition, Prentice Hall International, 1992.

STA87. The STARTS Guide (Standard Descriptions), 2nd Edition, Volume 2, NCC Publications, 1987.

Ste90. Stepney, S., *Specifying entity relationship diagrams in Z*, ORCA/Logical, 1990.

Sul93. Sully, P., *Modelling the World with Objects*, 2nd edition, Prentice-Hall International, 1993.

USA73. High-ranking USAF decision maker, quote from lecture notes of Norcliffe, A., Sheffield Hallam University, quote dated 1973.

Val87. Valentine, S., *Why Z?*, Systems International (Software Specification), March, 1987.

Val91. Valentine, S., *Z--, an Executable Subset of Z*, in *Z User Workshop*, York 1991, Nicholls, J.E., Ed., Springer-Verlag, 1992.

Val94. Valentine, S., *The Z-- Language*, Internal Document, University of Brighton, March, 1994.

Wes92. West, M., M. and Eaglestone, B., M., *Software development: two approaches to animation of Z specifications using Prolog*, Software Engineering Journal, July, 1992.

Wir82. Wirth, N., *Program Development by Stepwise Refinement*, Comms. of the Association of Computing Machinery, 1971, reproduced in *Writings of the Revolution: Selected Readings on Software Engineering*, Yourdon, E., (Ed.), Yourdon Press, N.Y., 1982.

Woo94. Woodcock, J. C. P., Gardiner, P. H. B. and Hulance, J. R., *The Formal Specification in Z of Defence Standard 00-56*, Formal Systems (Europe) Ltd, 1994.

Wor92. Wordsworth, J. B., *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering*, Addison-Wesley, 1992.

Zav91. Zave, P. and Jackson, M., *Techniques for Partial Specification and Specification of Switching Systems*, Proceedings of the sixth annual Z User Meeting, Dec. 1991.

WRITING AND ANIMATING Z SPECIFICATIONS

Simon John Andrews

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

VOLUME OF APPENDICES

November 1996

LIST OF APPENDICES

APPENDIX A: PUBLISHED PAPERS

APPENDIX A.1: A CASE Tool for Demonstrating Z Specifications [And90]

CASE tool for demonstrating Z specifications

Andrews and A Norcliffe

he CASE tool we describe is designed to enable software engineers to produce a faithful nimation of specifications written in Z. Desirable properties which we feel animations of this 'nd should possess, and which have guided us in developing the tool, are the following.

. The executable code (ie the animation) must be easy to produce.

. The structure of the code should not be too far removed from the Z.

. The animation should be sufficiently user friendly to enable a client to understand and interact with it, thus facilitating the process of validating a specification against user requirements.

he CASE tool is based around the program development tool known as CRYSTAL. ᴙYSTAL is reasonably well-known in AI circles and is sold as an expert system shell by telligent Environments Ltd in Richmond. It is essentially a rule-based programming language ffering excellent input, output, and menu facilities, as well as all the standard features pected of any expert system shell. The specific advantages we see, that this environment ffers as a means of transforming Z to executable code, are as follows.

. The rule-based nature of CRYSTAL means that lines of Z, in the predicate of a schema, transform almost one-for-one into rules in CRYSTAL.

. The expandable way in which rules are built up in CRYSTAL mirrors very closely the use of the schema calculus in Z. The developer, using the tool, can faithfully transform a Z specification starting at the schema level and finishing at the line-by-line predicate level.

. The excellent user interface that comes with CRYSTAL enables the developer to concentrate his efforts on transforming Z instead of worrying about how to create a friendly user interface. This is an added bonus given the fact that implementation issues are positively avoided in formal specifications.

. The animation that results can be viewed by the client at different levels. This is possible because of the folded nature of the rule-based programming in CRYSTAL. At the highest level a system might be viewed as a menu having several options such as

```
quit
initialise state
save state
load state
print state
test data invariants
operation 1
operation 2
      .
      .
      .
operation n
```

Andrews and A Norcliffe are both members of the School of Engineering Information echnology at Sheffield City Polytechnic

Any operation chosen by the client can be systematically unfolded to discover the rules that make it work, thus promoting the vital interaction between client, developer and system that is necessary for requirements validation. In CRYSTAL this is feasible because at the highest level the rules are written in English. Only at the lowest level does English give way to code. What the client sees, therefore, is a faithful English translation of the developer's Z.

o illustrate these points a short example is now considered. The following is part of the Z pecification of a very simple security system that might be in operation in a building to onitor the whereabouts of staff users. The system state consists of three subsets, in, out and ers, of type P(STAFF_ID), and is represented by the following state schema

```
┌─State──────────────────────────
│ in, out, users : P(STAFF_ID)
├────────────────────
│ in ∩ out = {}
│ in ∪ out = users
└────────────────────────────────
```

mongst other things the system checks people in and out of the building and the CheckInOK peration may be specified as follows

```
┌─CheckInOK──────────────────────
│ Δ State
│ person_id? : STAFF_ID
├────────────────────
│ person_id? ∈ out
│ out'   = out \ {person_id?}
│ in'    = in ∪ {person_id?}
│ users' = users
└────────────────────────────────
```

When the precondition is violated the CheckInOK operation will fail. A robust CheckIn operation can therefore be defined as follows

CheckIn ≙ CheckInOK ∨ CheckInError

CheckInError ≙ CheckInError1 ∨ CheckInError2

where the two error schemas are as follows

```
┌─CheckInError1──────────────────────────
│ State
│ person_id? : STAFF_ID
│ message!   : REPORT
├────────────────────
│ person_id? ∈ in
│ message! = "Person already in building"
└────────────────────────────────────────
```

```
┌─CheckInError2──────────────────────────
│ State
│ person_id? : STAFF_ID
│ message!   : REPORT
├────────────────────
│ person_id? ∉ users
│ message! = "Person is not a valid user"
└────────────────────────────────────────
```

t the highest level the CRYSTAL coding for this Z could be the following

```
      CheckIn works
IF    CheckInOK works
OR    CheckInError works

      CheckInError works
IF    CheckInError1 applies
OR    CheckInError2 applies

      CheckInOK works
IF    person_id is entered into the system
AND   the person_id currently belongs to the set out
AND   the person_id is then removed from the set out
AND   the person_id is then added to the set in
AND   the set users is unchanged
AND   completion of the operation has been signalled

      CheckInError1 applies
IF    person_id is entered into the system
AND   the person_id currently belongs to the set in
AND   an appropriate message is output

      CheckInError2 applies
IF    person_id is entered into the system
AND   the person_id does not currently belong to the set users
AND   an appropriate message is printed
```

bviously the developer has to expand each of these individual rules further until they are pable of being executed. But in principle this is a fairly straightforward task given the vailable CRYSTAL operations, and the fact that sets, functions, relations, sequences, power ts, bags etc can all be represented conveniently as arrays in CRYSTAL.

'hat we observe, then, is that CRYSTAL rules are not too far removed from Z and give a ery faithful transformation of the Z. The animation at this level is also capable of being ıderstood by a client even though he may know little or no Z. The client can thus interact ith the specification through the CRYSTAL animation and can thus contribute meaningfully to e process of requirements validation.

'ith regard to the CASE tool, however, the following points are relevant. A major isadvantage of CRYSTAL is ~~that~~ although at a high level it faithfully represents the Z otation, at the lowest level the CRYSTAL code can be somewhat lengthy. For example, the RYSTAL transformation of a function override operation could require 50 or more lines of ode. This problem is further compounded by the fact that there is no parameter passing in RYSTAL ie it is not possible to write a single routine for ⊕ and pass the appropriate rameters to it. The code must be repeated each time it is required. However, this roblem of low-level coding can be avoided by writing a "Z-function" interface to CRYSTAL C and work is currently in progress to create a library of Z-function routines (⊕, ∪, ∩, # c). These will eventually be amalgamated with the standard CRYSTAL function library pplied with the shell and used in the same way in the CRYSTAL code. The result should a CASE tool that software engineers can use, with relative ease, to animate specifications itten in Z.

APPENDIX A.2: An Expert System CASE Tool for Simulating Z Specifications

[And91]

An expert system CASE tool for simulating Z specifications

Simon Andrews and Allan Norcliffe *

School of Engineering Information Technology
Sheffield City Polytechnic
Pond Street
Sheffield S1 1WB

Summary

In this paper we describe a prototype of an expert system support environment to assist software engineers in the development of information systems. The CASE tool is designed specifically to aid software engineers at the requirements capture and specification stage by providing a faithful simulation of specifications written in the formal notation Z. In the paper we consider the rationale for the tool and illustrate its essential features by showing how part of a Z specification may be implemented and therefore simulated using the tool. We look specifically at how the tool can be used by the developer to demonstrate the features of a system to a client and how the client, in turn, can interact with the specification without necessarily having to understand the Z notation used. Limitations of the tool are described together with some of the ways of overcoming these.

1. Background

Z is nothing more than a mathematical notation. It was developed originally by the Programming Research Group at Oxford to enable software engineers write formal specifications in a systematic way. As a language it has received much attention by both industry and academia and, along with VDM (Vienna Development Method), has become one of the standard languages for specifying secure and safety–critical systems. A handbook setting out the latest version of the language is provided by Spivey (1989), and examples illustrating the use of the notation in software development can be found in the specification casestudies book by Hayes (1987).

Formal methods, such as Z and VDM, offer one realistic way of combatting the software crisis, and their use in industry holds out the promise of improved software quality and better productivity of the software development process. By using formal methods the software engineer is able to produce a system specification that is precise and capable of being reasoned with mathematically. The specification can be shown to be internally consistent and any coding produced from it can be developed rigorously and systematically, and be shown to match the specification. The full force of mathematical analysis and proof can be brought to bear and this is what makes the use of formal methods so attractive. Formal methods therefore lead to verifiable code and thus enable the software engineer to build systems correctly.

2. The Rationale for the tool

Having the power to build software systems correctly throws into sharp focus the vital need to capture the user's requirements correctly at the specification stage. Building a system correctly using formal methods does not guarantee that the resulting system is the correct system for the user.

This state of affairs is not new, of course, and is one of the ever present problems that modellers have to face up to when solving problems using mathematics. Once formulated in terms of mathematics, a problem can in principle be solved correctly using mathematical techniques. The solution, however, is only as good as the assumptions behind the original model and exactly the same is true in software development. Once a formal specification has been drawn up, correct coding can certainly be produced, but this coding is only as good as the assumptions underpinning the original specification. If the assumptions are not what the user intended or wanted, then the system is in many ways invalid.

Validating specifications against user requirements is therefore a major problem in software engineering, and it is one for which formal methods do not readily provide an easy answer. It is hardly feasible to ask the user or client if the Z specification of their system is a valid interpretation of their requirements. Mathematics at the moment, unfortunately, is still for the initiated and is most likely to remain so. Somehow the software engineer has to be able to demonstrate the specification to the customer more directly in order to validate it against their requirements.

There are basically two ways in which a software engineer can demonstrate a specification. He could obviously specify the system in a language or notation that executes. PROLOG, OBJ or ML, for example, are such langauges. By executing the specification the customer requirements can be validated first hand by seeing the system in action. The second way is to develop the specification first in a language, such as Z, that does not execute, and then transform it to one that will. This is the essence of rapid prototyping. From the behaviour of the prototype, providing it is faithful to the original specification, the customer can then see if the requirements are being met. Rapid prototyping has the advantage of giving the developer more freedom at the specification stage since he is not constrained by working only in terms of a langauge that is capable of execution. In many ways mathematics is more expressive than most executable specification langauges, and rapid prototyping is often the preferred route.

The CASE tool that we now describe is a tool for validating a specification against user requirements via this second approach. With the CASE tool a simulation of the specification is provided which is faithful to the original Z and which animates or simulates the ideas captured in the specification.

3. The tool in outline

Desirable properties which we feel animations, or simulations, of this kind should possess, and which have guided us in developing the tool, are the following.

1. The executable code (ie the animation) must be easy to produce.

2. The structure of the code should not be too far removed from the Z.

3. The animation should be sufficiently user friendly to enable a client to understand and interact with it, thus facilitating the process of validating a specification against user requirements.

The CASE tool is based around the program development tool known as CRYSTAL. CRYSTAL is reasonably well-known in AI circles and is sold as an expert system shell by Intelligent Environments Ltd in Richmond. It is essentially a rule-based programming language offering excellent input, output, and menu facilities, as well as all the standard features expected of any expert system shell. The specific advantages we see, that this environment offers as a means of transforming Z to executable code, are as follows.

1. The rule-based nature of CRYSTAL means that lines of Z, in the predicate of a schema, transform almost one-for-one into rules in CRYSTAL.

2. The expandable way in which rules are built up in CRYSTAL mirrors very closely the use of the schema calculus in Z. The developer, using the tool, can faithfully transform a Z specification starting at the schema level and finishing at the line-by-line predicate level.

3. The excellent user interface that comes with CRYSTAL enables the developer to concentrate his efforts on transforming Z instead of worrying about how to create a friendly user interface. This is an added bonus given the fact that implementation issues are positively avoided in formal specifications.

4. The animation that results can be viewed by the client at different levels. This is possible because of the folded nature of the rule-based programming in CRYSTAL. At the highest level a system might be viewed as a menu having several options such as

```
quit
initialise state
save state
load state
print state
test data invariants
operation 1
operation 2
      .
      .
      .
operation n
```

Any operation chosen by the client can be systematically unfolded to discover the rules that make it work, thus promoting the vital interaction between client, developer and system that is necessary for requirements validation. In CRYSTAL this is feasible because at the highest level the rules are written in English. Only at the lowest level does English give way to code. What the client sees, therefore, is a faithful English translation of the developer's Z.

4. <u>Using the tool</u>

To illustrate these points a short example is now considered. The following is part of the Z specification of a very simple security system that might be in operation in a building to monitor the whereabouts of staff users. The system state consists of three subsets, in, out and users, of type P(STAFF_ID), and is represented by the following state schema

```
┌─State──────────────────────────
│ in, out, users : P(STAFF_ID)
│
├────────────────────
│
│ in ∩ out = {}
│ in ∪ out = users
│
└────────────────────────────────
```

Amongst other things the system checks people in and out of the building and the

CheckInOK operation may be specified as follows

```
┌─CheckInOK──────────────────
│ Δ State
│ person_id? : STAFF_ID
├──────────────────
│ person_id? ∈ out
│ out'   = out \ {person_id?}
│ in'    = in ∪ {person_id?}
│ users' = users
└────────────────────────────
```

When the precondition is violated the CheckInOK operation will fail. A robust CheckIn operation can therefore be defined as follows

$\text{RCheckIn} \mathrel{\hat{=}} (\text{CheckIn} \wedge \text{Success}) \vee \text{CheckInError}$

$\text{CheckInError} \mathrel{\hat{=}} \text{StaffIn} \vee \text{NotUser}$

where the success and two error schemas are as follows

```
┌─Success────────────────────
│ result! : REPORT
├──────────────────
│ result! = ok
└──────────────────────────────────
```

```
┌─StaffIn────────────────────
│ Ξ State
│ person_id? : STAFF_ID
│ result!    : REPORT
├──────────────────
│ person_id? ∉ in
│ result! = already_in
└──────────────────────────────────
```

```
┌─NotUser────────────────────
│ Ξ State
│ person_id? : STAFF_ID
│ result!    : REPORT
├──────────────────
│ person_id? ∉ users
│ result! = not_known
└──────────────────────────────────
```

At the highest level the CRYSTAL coding for this Z could be the following

```
        RCheckIn works
IF      CheckIn works
AND     Success is indicated
OR      CheckInError works

        CheckInError works
IF      StaffIn applies
OR      NotUser applies
```

At the next level down these rules might be expanded as follows

```
        CheckIn works
IF      person_id is entered into the system
AND     the person_id currently belongs to the set out
AND     the person_id is then removed from the set out
AND     the person_id is then added to the set in
AND     the set users is unchanged

        Success is indicated
IF      the result "ok" is output

        StaffIn applies
IF      person_id is entered into the system
AND     the person_id currently belongs to the set in
AND     the result "already in" is output

        NotUser applies
IF      person_id is entered into the system
AND     the person_id does not currently belong to the set users
AND     the result "not known" is output
```

Obviously the developer has to expand each of these individual rules further until they are capable of being executed. But in principle this is a fairly straightforward task given the available CRYSTAL operations, and the fact that sets, functions, relations, sequences, power sets, bags etc can all be represented conveniently as arrays in CRYSTAL

5. Advantages of the tool

Most of these have been listed previously, but it is worth pointing out the advantages again.

(1) We observe that the CRYSTAL is very faithful to the Z. The simulation that is produced when the CRYSTAL code is executed is indeed a simulation of the specification and not an implementation that is far removed from the Z.

(2) Since the CRYSTAL mirrors the structure of the Z so closely it is a relatively easy task for the developer to begin the process of developing the executable code. The excellent user interface that comes with CRYSTAL is very helpful when lower-level coding has to be developed.

(3) The high-level coding, being written in English, is clearly capable of being understood by a client even though he may know little or no Z. The English translation of the Z in CRYSTAL does not introduce potentially harmful ambiguities and via this translation the client can thus interact with

the specification and contribute meaningfully to the process of requirements validation.

(4) The three-way communication between customer, developer, and system, so vital for validation purposes, is thus possible via the tool.

6. Limitations of the tool and suggested improvements

The tool does have its limitations. A major disadvantage of CRYSTAL is that although at a high level it faithfully represents the Z notation, at the lowest level the CRYSTAL code can be somewhat lengthy. For example, the CRYSTAL transformation of a function override operation could require upwards of 50 lines of coding. This problem is further compounded by the fact that there is no parameter passing in CRYSTAL, ie it is not possible to write a single routine for function override, for example, and pass the appropriate parameters to it. The code must be repeated each time it is required.

However, this problem of low-level coding can be avoided by writing a "Z-function" interface to CRYSTAL in C and work is currently in progress to create a library of Z-function routines for the operations $\oplus$, $\cup$, $\cap$, #, \ etc. These will be eventually amalgamated with the standard CRYSTAL function library supplied with the shell and used in the same way in the CRYSTAL code.

7. Discussion

It is perhaps important to point out that the tool we have described is not yet commercially available, and indeed a full-scale prototype has not yet been built. The tool is still very much the subject of final year student projects, and only parts of it currently exist. The CASE tool proper, when it is finally built, will possess the following features.

(1) It will be built around the CRYSTAL environment for ease of use on a PC.

(2) There will be an appropriate additional library of Z functions to enable many standard Z operations to be carried out with few key strokes.

(3) It will contain an expert adviser on writing Z for various generic system types such as information systems.

(4) There will be help screens for transforming Z.

(5) There will be standard macros for creating system menus, input and output screens, initialising system states etc.

Given these features we believe the result will be a CASE tool that software engineers can use with relative ease to animate specifications written in Z.

References

1. HAYES, I. (Ed). (1987). Specification Case Studies, Prentice Hall.

2. SPIVEY, J.M. (1989). The Z Notation – A Reference Manual, Prentice Hall.

A LIFT SYSTEM

Although the Z notation is based on discrete mathematics, it can be used to model continuous systems if it is possible to define the evolution of the system in terms of discrete steps.

A lift system is such a system, where a lift moving from one floor to the next can be thought of as a discrete step.

The following describes the specification of a general lift system with *m* floors and *n* lifts.

System requirements

The requirements of the system are as follows.

- Each lift has a set of buttons, one for each floor. When pressed, a floor-request button will cause the lift to visit that floor.
- Each floor, except the ground and top floors, has two request buttons, one to request an up-lift and one to request a down-lift. At the ground floor only an up-lift can be requested, and at the top floor only a down-lift can be requested. The request is serviced when a lift visits the floor and is either moving in the desired direction or has no outstanding requests. In the latter case, if both floor buttons are pressed, only one should be cancelled. Waiting time for floors should be minimised.
- When a lift has no requests to service, it should remain in a holding position.
- All requests from floors must be serviced eventually, with all floors given equal priority.
- All requests from within a lift must be serviced eventually, with floors being serviced sequentially in the direction of travel.
- Each lift has an emergency button which puts it out of service, and each lift has a mechanism to cancel its out of service status.

e system state

e lift system has m floors and n lifts. We define m and n to be global ·ables as follows.

$m, n : \mathbb{N}_1$

et of floors, and a set of lifts, can then be defined in terms of m and n in following way.

$FLOOR == 1..m$
$LIFT == 1..n$

ch lift can be in one of four states: moving up, moving down, holding, or t of service. We thus define a free type, *DIRECTION*, as follows.

$DIRECTION ::= up \mid down \mid holding \mid out$

e status of all lifts can now be represented by a total function:

$dir : LIFT \rightarrow DIRECTION$

early, it is also important to know on which floor a lift currently is:

$floor\_on : LIFT \rightarrow FLOOR$

moving lift is deemed to be at the floor it has just passed until it arrives or passes, the next floor. Requests for lifts to stop at particular floors of three types: requests made within lifts, requests made on floors for wn-lifts, and requests made on floors for up-lifts. The requests made m within the lifts are modelled by the following total function.

$reqs\_in\_lift : LIFT \rightarrow \mathbb{F}\, FLOOR$

e requests made on floors are modelled by finite sets alone:

$reqs\_dn : \mathbb{F}\, FLOOR$
$reqs\_up : \mathbb{F}\, FLOOR$

e now have all the components of the system apart from the fact that a quest to go up on the top floor, and a request to go down on the ground floor, e impossible.

$reqs\_up \subseteq 1 .. m-1$
$reqs\_dn \subseteq 2 .. m$

The state schema can now be written down as follows.

LiftState

$dir : LIFT \rightarrow DIRECTION$
$floor\_on : LIFT \rightarrow FLOOR$
$reqs\_dn : \mathbb{F}\, FLOOR$
$reqs\_up : \mathbb{F}\, FLOOR$
$reqs\_in\_lift : LIFT \rightarrow \mathbb{F}\, FLOOR$

$reqs\_up \subseteq 1 \mathrel{..} m-1$
$reqs\_dn \subseteq 2 \mathrel{..} m$

The initial state

It is sometimes helpful to give an example of a valid starting state. One choice is the state where all the lifts are in a holding position on the ground floor, and there are no requests in the system.

InitState

LiftState

$dir = \{\, l : LIFT \bullet l \mapsto holding \,\}$
$floor\_on = \{\, l : LIFT \bullet l \mapsto 1 \,\}$
$reqs\_dn = \varnothing$
$reqs\_up = \varnothing$
$reqs\_in\_lift = \varnothing$

The actions of the lift system

Now we have the state, we must consider the actions of the system. They are listed below.

- *StartDown* - describes the conditions needed for lifts to start moving downwards from a holding position.

- *StartUp* - describes the conditions needed for lifts to start moving upwards from a holding position.

• *MoveDown* -	describes downward lifts moving down a floor.
• *MoveUp* -	describes upward lifts moving up a floor.
• *HoldDown* -	describes downward lifts remaining in a holding position.
• *HoldUp* -	describes upward lifts remaining in a holding position.
• *ReqInLift* -	describes requests being made from within lifts.
• *ReqUp* -	describes requests to go up being made on floors.
• *ReqDown* -	describes requests to go down being made on floors.
• *Stop*$_x$ -	describes lifts servicing floors (halting if necessary, opening the lift doors, waiting a pre-set length of time, and then closing the doors). There are several cases of the *Stop* action.
• *OutOfService* -	describes lifts going out of service.
• *BackInService* -	describes lifts coming back into service.

e consider each of these system actions in turn.

e *StartDown* and *StartUp* actions

lift will start to move downwards from a holding position if

- there is a request for a lower floor made from within the lift, or
- there is a request to go down made on a lower floor, and
    - there is no holding lift above the floor and closer, and
    - there is no downward lift above the request floor, and at the same level or below the lift, or

- there is a request to go up made on a lower floor, and
    - there is no upward or holding lift below the floor, and
    - there is no holding lift above the floor and closer.

This condition is complex but essential. The condition for a lift to start downwards (or upwards) is at the heart of the system. The lift in the best position should be selected to answer a particular request. Although it may be possible for there to be two or more lifts answering a particular request, each request should be answered in the fastest way possible, and no request can remain unanswered. If the above condition is met for a particular lift, then its direction will be changed from *holding* to *down*.

The actual *StartDown* action that causes holding lifts to start to move down can be specified as follows.

$$\begin{array}{|l}
\textit{StartDown} \\
\hline
\Delta \textit{LiftState} \\
\hline
dir' = dir \oplus \{lift : LIFT \mid dir(lift) = holding \wedge \\
\qquad \exists f : FLOOR \bullet ((f < floor\_on(lift)) \wedge (f \in reqs\_in\_lift)) \vee \\
\qquad\qquad (f \in reqs\_dn \wedge \neg \exists\, l : LIFT \bullet ((dir(l) = holding \wedge \\
\qquad\qquad f < floor\_on(l) < floor\_on(lift)) \vee \\
\qquad\qquad (dir(l) = down \wedge f < floor\_on(l) \leq floor\_on(lift)))) \vee \\
\qquad\qquad (f \in reqs\_up \wedge \neg \exists\, l : LIFT \bullet (((dir(l) = up \vee \\
\qquad\qquad dir(l) = holding) \wedge floor\_on < f) \vee \\
\qquad\qquad (dir(l) = holding \wedge f < floor\_on(l) < floor\_on(lift))))) \\
\qquad \bullet\ lift \mapsto down\} \\
reqs\_dn' = reqs\_dn \\
reqs\_up' = reqs\_up \\
reqs\_in\_lift' = reqs\_in\_lift \\
floor\_on' = floor\_on \\
\hline
\end{array}$$

The *StartUp* action is clearly going to take a very similar form (a sort of mirror image in fact) and in the interest of brevity it is not given here.

Technical Note

The *StartDown* schema has no precondition (except the data invariant) and will always apply. If there are no holding lifts satisfying the predicate in the above set comprehension the value of *dir* is left unchanged, along with the other state variables. The comprehension is thus empty. If there are holding lifts satisfying the predicate in the comprehension, their directions will be changed to *down* and the function *dir* is updated accordingly. Either way the *StartDown* schema applies.

The *StartUp* schema when written out will similarly have no precondition and it, too, will always apply.

End of technical note

he *MoveDown* and *MoveUp* actions

ese actions are straightforward. If the direction of a lift is *down*, then it will ove down one floor; if the direction is *up*, it will move up one floor. The *oveDown* and *MoveUp* actions are designed to follow the *StartDown* and *tartUp* actions. Their specification is as follows.

MoveDown

$\Delta LiftState$

$$\begin{array}{l}
floor\_on' = floor\_on \oplus \{ \, lift : LIFT \mid dir(lift) = down \\
\qquad\qquad\qquad\qquad \bullet \; lift \mapsto (floor\_on(lift) - 1) \, \} \\
dir' = dir \\
reqs\_dn' = reqs\_dn \\
reqs\_up' = reqs\_up \\
reqs\_in\_lift' = reqs\_in\_lift
\end{array}$$

MoveUp

$\Delta LiftState$

$$\begin{array}{l}
floor\_on' = floor\_on \oplus \{ \, lift : LIFT \mid dir(lift) = up \\
\qquad\qquad\qquad\qquad \bullet \; lift \mapsto (floor\_on(lift) + 1) \, \} \\
dir' = dir \\
reqs\_dn' = reqs\_dn \\
reqs\_up' = reqs\_up \\
reqs\_in\_lift' = reqs\_in\_lift
\end{array}$$

The *HoldDown* and *HoldUp* actions

The conditions for a downward lift to remain in a holding position are

- its direction is *down*
- there are no requests in the lift for a lower floor
- there is no downward request on a lower floor
- there is no upward request on a lower floor.

Again, these conditions ensure that all requests will be serviced even if it means that two or more lifts are "racing" to service a particular request. The conditions also ensure that a downward lift will hold once it has reached the ground floor (and not continue downward into the foundations!). If the conditions are all satisfied a lift's direction is changed to *holding*.

The *HoldDown* schema can be written as follows.

$$
\begin{array}{|l}
\textit{HoldDown} \\
\hline
\Delta \textit{LiftState} \\
\hline
dir' = dir \oplus \{\, lift : LIFT \mid dir(lift) = down \land \\
\qquad \neg \exists f : FLOOR \bullet (f < floor\_on(lift) \land \\
\qquad\qquad (f \in reqs\_dn \lor f \in reqs\_up \lor \\
\qquad\qquad f \in reqs\_in\_lift(lift))) \\
\qquad \bullet\ lift \mapsto holding \,\} \\
\\
floor\_on' = floor\_on \\
reqs\_dn' = reqs\_dn \\
reqs\_up' = reqs\_up \\
reqs\_in\_lift' = reqs\_in\_lift \\
\hline
\end{array}
$$

The *HoldUp* action can be specified in a very similar mirror-image form, and for reasons of brevity is not given here.

The *ReqInLift* action

For a request to be made from within a lift we simply specify that a pair of inputs must exist: a floor number and a lift number. The lift must also not be out of service. The floor number is then added to the function *reqs_in_lift* for that lift.

e schema that specifies *ReqInLift* is the following one.

$$\begin{array}{|l}
\hline
\textit{ReqInLift} \\
\Delta \textit{LiftState} \\
\textit{reqs?} : \mathbb{P}(\textit{LIFT} \times \textit{FLOOR}) \\
\hline
\textit{reqs\_in\_lift}' = \textit{reqs\_in\_lift} \oplus \\
\qquad \{ l : \textit{LIFT} \mid l \in \operatorname{dom} \textit{reqs?} \wedge \textit{dir}(l) \neq \textit{out} \\
\qquad\quad \bullet\ l \mapsto (\textit{reqs\_in\_lift}(l) \cup \operatorname{ran}(\textit{reqs?}))\} \\
\textit{dir}' = \textit{dir} \\
\textit{floor\_on}' = \textit{floor\_on} \\
\textit{reqs\_dn}' = \textit{reqs\_dn} \\
\textit{reqs\_up}' = \textit{reqs\_up} \\
\hline
\end{array}$$

e *ReqUp* and *ReqDown* actions

ese actions are even more straightforward than *ReqInLift*. All that is quired is that an input of a floor number exists. If that is the case, the floor nber is added to the requests to go down in *ReqDown* or added to the quests to go up in *ReqUp*. The actions are very similar, so only *ReqDown* given here.

$$\begin{array}{|l}
\hline
\textit{ReqDown} \\
\Delta \textit{LiftState} \\
\textit{floors?} : \mathbb{P}(\textit{FLOOR}) \\
\hline
\textit{reqs\_dn}' = \textit{reqs\_dn} \cup \\
\qquad \{ f : \textit{FLOOR} \mid f \in \textit{floors?} \wedge f \neq 0 \} \\
\textit{dir}' = \textit{dir} \\
\textit{floor\_on}' = \textit{floor\_on} \\
\textit{reqs\_up}' = \textit{reqs\_up} \\
\textit{reqs\_in\_lift}' = \textit{reqs\_in\_lift} \\
\hline
\end{array}$$

ıe first of the schema predicates requires that $f \neq 0$. This is because there no down button on the ground floor. Similarly in *ReqUp*, $f \neq m$ ensures an p request cannot be made from the top floor.

single request action, dealing with all requests, can now be formed as llows.

$$\textit{Reqs} \mathrel{\widehat{=}} \textit{ReqInLift} \fatsemi \textit{ReqDown} \fatsemi \textit{ReqUp}$$

The $Stop_x$ actions

There are six cases we need to consider that describe lifts servicing floors.

- $Stop_1$ - servicing floor requests made from within lifts
- $Stop_2$ - servicing down requests whilst travelling downwards
- $Stop_3$ - servicing up requests whilst travelling upwards
- $Stop_4$ - servicing down requests whilst travelling upwards (this caters for the top floor)
- $Stop_5$ - servicing up requests whilst travelling downwards (this caters for the ground floor)
- $Stop_6$ - servicing floors by holding lifts where there is a request down and/or a request up on that floor.

1. The $Stop_1$ action

In this first case, of lifts servicing a floor request made from within the lift, the conditions that must be satisfied are

- the lift is not out of service
- there is a request within the lift for the floor on which the lift is

If these are satisfied then it is assumed that the floor is serviced. The request is removed from the set of requests for that lift. The schema for $Stop_1$ is as follows.

$$
\begin{array}{|l}
\hline
\textit{Stop}_1 \\
\Delta \textit{LiftState} \\
\hline
\textit{reqs\_in\_lift}' = \textit{reqs\_in\_lift} \oplus \{\, \textit{lift} : \textit{LIFT} \mid \\
\qquad \textit{dir}(\textit{lift}) \neq \textit{out} \land \\
\qquad \textit{floor\_on}(\textit{lift}) \in \textit{reqs\_in\_lift}(\textit{lift}) \\
\qquad \bullet\ \textit{lift} \mapsto \textit{reqs\_in\_lift}(\textit{lift}) \setminus \{\textit{floor\_on}(\textit{lift})\}\} \\
\textit{dir}' = \textit{dir} \\
\textit{floor\_on}' = \textit{floor\_on} \\
\textit{reqs\_dn}' = \textit{reqs\_dn} \\
\textit{reqs\_up}' = \textit{reqs\_up} \\
\hline
\end{array}
$$

The *Stop*$_2$ action

the *Stop*$_2$ case, of lifts servicing down requests whilst travelling nwards, the conditions to be satisfied by a lift are

- the lift is travelling downwards
- there is a down request on the floor at which the lift is

ıese conditions are satisfied then the floor will be serviced. The request removed from the set of down requests.

```
───── Stop₂ ──────────────────────────
│ ΔLiftState
├──────────────────────
│ reqs_dn' = reqs_dn \ { lift : LIFT |
│                 dir ( lift ) = down ∧
│                 floor_on ( lift ) ∈ reqs_dn • floor_on ( lift ) }
│ dir' = dir
│ floor_on' = floor_on
│ reqs_up' = reqs_up
│ reqs_in_lift' = reqs_in_lift
└──────────────────────────────────────
```

The *Stop*$_3$ action

ıs is the opposite of *Stop*$_2$. A lift's direction must be upwards instead of wnwards, and the request is to go up rather than down. For brevity we do t give the specification of the action here.

The *Stop*$_4$ action

e *Stop*$_4$ action, of lifts servicing down requests whilst travelling upwards, a little more involved. The conditions to be satisfied by each lift are that

- the lift is travelling upwards
- there is a request to go downwards on the floor at which the lift is
- there are no requests within the lift for a higher floor
- there are no requests up which are on or above the floor
- there are no requests down above the floor

If these hold then the floor is serviced and the request removed. Lifts are set to a holding position to allow them to change direction.

$$\begin{array}{|l}
\hline
\quad Stop_4 \\
\Delta LiftState \\
\hline
reqs\_dn' = reqs\_dn \setminus \{\, lift : LIFT \mid dir(lift) = up \wedge floor\_on(lift) \in reqs\_dn \wedge \\
\qquad \neg \exists f : FLOOR \bullet ((f > floor\_on(lift) \wedge (f \in reqs\_in\_lift(lift) \vee \\
\qquad\quad f \in reqs\_dn)) \vee (f \geq floor\_on\ (lift) \wedge f \in reqs\_up)) \\
\qquad\quad \bullet floor\_on(lift)\} \\
dir' = dir \oplus \{\, lift : LIFT \mid dir(lift) = up \wedge floor\_on(lift) \in reqs\_dn \wedge \\
\qquad \neg \exists f : FLOOR \bullet ((f > floor\_on(lift) \wedge (f \in reqs\_in\_lift(lift) \vee \\
\qquad\quad f \in reqs\_dn)) \vee (f \geq floor\_on\ (lift) \wedge f \in reqs\_up)) \\
\qquad\quad \bullet lift \mapsto holding\} \\
floor\_on' = floor\_on \\
reqs\_up' = reqs\_up \\
reqs\_in\_lift' = reqs\_in\_lift \\
\hline
\end{array}$$

5. The *Stop*$_5$ action

The *Stop*$_5$ action is the "mirror image" of *Stop*$_4$. Again, for reasons of brevity, we do not specify this action here.

6. The *Stop*$_6$ action

Lastly, the *Stop*$_6$ action is more complicated since it must cater for requests up and/or requests down.

A holding lift can service an up or down request on its floor. Where both requests exist, the down request is chosen if the lift is nearer to the bottom than the top, whilst the up request is chosen if it is nearer the top.

The necessary conditions are that for a holding lift to service a down request either there is no up request on the same floor, or there is also an up request but the floor is on or below halfway. This halfway constraint will be incorporated into the specification of *Stop*$_6$ by requiring that $2*floor\_on(lift) \leq m$.

To service an up request there must either be no down request, or there is also a down request but the floor is above halfway. This halfway constraint will be incorporated by requiring that $2*floor\_on(lift) > m$.

The appropriate specific~~ation~~ of *Stop*$_6$, that encapsulates these conditions, and contains the appropriate system updates may be written as follows.

$Stop_6$

$\Delta LiftState$

$reqs\_dn' = reqs\_dn \setminus \{ lift : LIFT \mid dir(lift) = holding \wedge$
$((floor\_on(lift) \in reqs\_dn \wedge floor\_on(lift) \notin reqs\_up) \vee$
$((floor\_on(lift) \in (reqs\_dn \cap reqs\_up)) \wedge$
$2 * floor\_on(lift) \leq m)) \bullet floor\_on(lift)\}$

$reqs\_up' = reqs\_up \setminus \{ lift : LIFT \mid dir(lift) = holding \wedge$
$((floor\_on(lift) \in reqs\_up \wedge floor\_on(lift) \notin reqs\_dn) \vee$
$(floor\_on(lift) \in (reqs\_up \cap reqs\_dn)) \wedge$
$2 * floor\_on(lift) > m)) \bullet floor\_on(lift)\}$

$dir' = dir$
$reqs\_in\_lift' = reqs\_in\_lift$
$floor\_on' = floor\_on$

of the above six stopping actions can be combined in one *Stop* action ing schema disjunction and schema composition as follows.

$$Stop \mathrel{\widehat=} Stop_1 \mathbin{⨾} (Stop_2 \vee Stop_3 \vee Stop_4 \vee Stop_5 \vee Stop_6)$$

hema composition is used following $Stop_1$ because postconditions volving *reqs_in_lift'*, *reqs_dn'* and *reqs_up'* would clash if a lift ıultaneously serviced a floor request from within the lift and a request ade on a floor.

e *OutOfService* and *BackInService* actions

ese two simple actions complete the action set. They both require that an put of a lift exists. Both schemas corresponding to these actions are given low.

ote that when a lift goes out of service, any requests that have been made 'thin the lift are lost.

ts that have been out of service are brought back into service via the *ackInService* action.

OutOfService

$$\begin{array}{l}
\Delta LiftState \\
lifts? : \mathbb{P}(LIFT) \\
\hline
dir' = dir \oplus \{ lifts? \mapsto out \} \\
reqs\_dn' = reqs\_dn \\
reqs\_up' = reqs\_up \\
floor\_on' = floor\_on \\
reqs\_in\_lift' = reqs\_in\_lift \oplus \{ lift : LIFT \mid \exists\, l? = lift \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \bullet lift \mapsto \varnothing \}
\end{array}$$

BackInService

$$\begin{array}{l}
\Delta LiftState \\
lifts? : \mathbb{P}(LIFT) \\
\hline
dir' = dir \oplus \{ lift : LIFT \mid \\
\qquad\qquad lift \in lifts? \wedge dir(lift) = out \\
\qquad\qquad \bullet\ lift \mapsto holding \} \\
floor\_on' = floor\_on \\
reqs\_up' = reqs\_up \\
reqs\_dn' = reqs\_dn \\
reqs\_in\_lift' = reqs\_in\_lift
\end{array}$$

Lifts that are put back into service are therefore set to a holding position.

The Complete System

A single discrete step can now be formed as follows.

$$\begin{array}{ll}
Step \;\widehat{=} & Reqs \fatsemi Stop \fatsemi Reqs \fatsemi HoldDown \fatsemi Reqs \fatsemi \\
& HoldUp \fatsemi Reqs \fatsemi Startdown \fatsemi Reqs \fatsemi StartUp \fatsemi \\
& Reqs \fatsemi MoveDown \fatsemi Reqs \fatsemi MoveUp \fatsemi Reqs \fatsemi \\
& OutOfService \fatsemi Reqs \fatsemi BackInService
\end{array}$$

The *Reqs* action is used at every opportunity to try to simulate the fact that requests can be made at any time.

ally, let us assume that the system has a fixed life span in terms of a mber of steps. If we define $k : \mathbb{N}_1$, such that k is the life span of the tem, we can define the complete system by writing

$$Lift \mathrel{\widehat{=}} InitState \mathbin{⨾} Step^k$$

's completes the specification of the lift system.

ritten by Simon Andrews, Aug 1990, Oct 1990, Nov 1990, May 1991.

Case Study No. 16

Title: *Learning contracts*

Authors: *Simon Andrews*

Department: *School of Engineering Information Technology*

Institution: *Sheffield Hallam University*

Characteristics of the Teaching & Learning Strategy		The Problems Tackled		Transferable Skills and Competences Developed	
Project	✓	Increasing group size		Initiative	✓
Laboratory	✓	Different Student ability levels	✓	Independence	
Fieldwork		Non-specialist subject		Teamwork	✓
Feedback to Student	✓	Part-time student	✓	Management/leadership	
Independent Study		Relevance/realism	✓	Time Management	✓
Ownership	✓	Teaching overload		Planning and Organising	✓
CAL		Staff assessment overload		Written Communication	✓
Group/Teamwork	✓	Student assessment overload		Oral Communication	
Written/Presentation	✓	Financial resources		Finding information	
Software Presentation	✓	Human resources		Data analysis	
Poster Presentation		Physical resources		Decision making	✓
Video Presentation		Quality of contact time	✓	Information Technology	✓
External Links		Collusion in Coursework		Problem solving	
Peer/self assessment		Validity of Coursework			
Flexible Tutorials					
Team Teaching					
Self-help groups					

Learning contracts

Simon Andrews
School of Engineering Information Technology
Sheffield Hallam University

Rationale

It has long been noted (1) that traditional mathematics courses do little to help the student acquire the wider transferable skills now needed by employers (see also 2, 3) - skills such as communication and negotiation, problem formulation, team work, and meeting deadlines. The use of learning contracts in the teaching of mathematics is one method that enables such skills to be developed and nurtured.

In industry contracts are drawn up for several good reasons. A contract, once negotiated, is a formal statement of what is to be delivered by the contractor, and an important reference to both contractor and contractee during the lifetime of the contract. A contract allows a complex task to be broken down into well defined stages with expected completion dates and a framework within which progress can be monitored and assessed.

In Higher Education these self same benefits are equally relevant to learning via project work, and especially so in the context of acquiring mathematical modelling skills through mini-projects or casestudies. A contract to learn pinpoints the purpose and aims of a project. A contract is a formal agreement encouraging responsibility within the student - the project is taken more seriously by the student. A structured contract gives the student a clear step-by-step approach to achieving a goal and stages break down a complex project into more easily managed tasks with fixed deadlines to meet and with clearly defined deliverables at each stage.

Negotiation of the details of the learning contract allows the student greater participation. If the student is allowed to determine, to some extent, the content that is needed to fulfil the learning aims of a project, then the student will be better motivated. The student should be given some say in the chosen vehicle or framework for learning, especially if a particular interest of a student is suitable around which to build a contract.

Having student input into the drawing up of a contract, in choosing a system to model, in deciding on a set of tasks to undertake (where choices are available), and in negotiating weightings for the tasks for assessment purposes, can imbue the student with a strong sense of ownership of the work to be undertaken. Student ownership should be intrinsic in the use of learning contracts; it is a powerful motivational force encouraging work of value and quality.

Description

Learning contracts are not new and have been used for several years on business studies courses. Their use in the teaching of mathematics, however, is less widespread. At Sheffield Hallam University learning contracts are used in the teaching of formal methods on the School of Engineering Information Technology's Masters programme in Engineering Information Technology (4). Formal methods are mathematically-based methods for developing software and concern the use of discrete mathematics and first-order predicate logic in the modelling of user requirements for software systems. Formal methods are now taught on a growing number of mathematics degrees in the UK, and an introduction to the use of discrete mathematics in software development can be found in several recent texts on the subject (5, 6, 7, 8).

After a period of conventional lecture, tutorial and laboratory work on formal methods, the MSc students at Sheffield Hallam are in a position to undertake a substantial mini-project or casestudy, either individually or in groups (this largely depends on class size) and are introduced to the idea of using a learning contract as a vehicle for undertaking and being assessed in a case study.

The lecturer, acting as contractee, negotiates a learning contract with each group or individual using a contract proforma. The proforma has a section for the casestudy title, the aims of the learning contract, a work plan and schedule for casestudy stages and tasks, a flexibility weighting (see later), special requirements and signatures of contractee and contractor/s. Although suggestions of suitable systems to model are given, the contractors are encouraged to suggest systems that they might be interested in modelling. They may be familiar, for example, with a particular system - part-time students are often working in a technical capacity and are usually keen to apply formal methods to their field. Examples of systems suggested by contractors and used successfully as the problem domain of learning contracts are:

- An Automated Teller Machine;
- An Electronic Components Thermal Evaluation System;
- An Intelligent Multi-storey Car Park;
- A Unix Style Process Scheduler;
- A Motorist's Route Planner;
- A British Railway's Rail Failure Database;
- A Warehouse Stock Control System.

The stages of the work plan are then agreed between contractee and contractor. The first two stages are invariably the same: the first being the writing of a requirements document for the system chosen, typically consisting of an informal description of the system and the data structures and operations to be performed on or by the system and will often contain diagrammatical representation such as flow charts and data flow diagrams; the second being the production of a documented abstract formal specification (mathematical model) of the functionality of the system. The contractor is then more or less free to pick and choose from various tasks that can be performed once an abstract formal specification has been formulated. These are to:

- refine the abstract specification to a more concrete specification (in fact, there is scope for two or even more stages of refinement);
- construct proofs of consistency within the formal specification;
- animate the specification (a form of rapid prototyping where the formal, mathematical structure of the specification remains intact), using a suitable AI language (PROLOG, CRYSTAL and KAPPA PC are examples);
- implement the formal specification in a target language such as C.

Typically the contractor is encouraged to select one or two of these additional tasks and, if two (or more) are chosen, it may be suggested to the contractor that only a part of the formal specification need be taken forward. If an animation or implementation is to be produced, then the contractor will be required to demonstrate it to the contractee as part of the assessment.

A mark for each stage is agreed upon along with the flexibility weighting. The flexibility weighting (typically 10-15%) is used by the contractor, in agreement with the contractee, to adjust the assessment weightings of the stages once the contract has been fulfilled. This adjustment may be acceptable if unforeseen problems or impracticalities arise at some stage, or if the contractor has been over ambitious (although ambition is not discouraged).

The final stage of the contract will always be to produce and submit a well documented report for assessment. A schedule (a due date for the completion of each stage of the contract) is also agreed between contractor and contractee. Special requirements, such as access to a particular programming language or computer workstation, are noted in the contract.

The aims of the contract can now be written into the contract in terms of the tasks to be undertaken, although the aims are inherently the same in each contract. Typically, the aims might be:

"To gain experience of formal specification and the techniques required to progress from a set of informal system requirements through an abstract formal specification to an animation of the specification."

Because of the use of a flexibility weighting the individual stages are assessed only after the casestudy report is handed in but adherence to the schedule can be taken into account. The stages are assessed with consideration given to the completeness of each stage and the contractor's ability to demonstrate that learning has taken place.

A workshop regime is used throughout the contract period, the contractee typically spending half an hour per week with each contractor. During this time the contractor plays two additional roles:

- that of the contractor of the system being developed, in discussing system requirements, constraints and boundaries;
- that of a mathematics consultant in formal specification, refinement and proof.

Evaluation

Learning contracts have, on the whole, proved to be very successful. The contractors work very hard to conform to the work plan and schedule that they have agreed. The quality of the work and the reports produced is consistently high and the quantity of work completed can be surprisingly large.

High flyers, given such a free hand in problem formulation, can produce outstanding pieces of work. It has been possible, to some extent, to tailor contracts to ability levels with, for example, less mathematically experienced students working on less complex or ambitious systems. The size and complexity of the system developed has then had some bearing on assessment.

Feedback from students has been universally positive. There have been comments like:

"It makes you feel you are doing something worthwhile and important",
"I like being in control" and
"Without a schedule to work to I would never have got this much done."

The students, in general, have worked seriously and enthusiastically and have made full and energetic use of their consultation periods in workshops. Several have produced systems that they feel will be useful or even have commercial possibilities! Many develop skills and an interest in the field to such an extent that they wish to pursue the subject further, either academically in projects or even research, or in terms of a career.

The main problem, from the contractor's point of view, has been undertaking an over ambitious contract, but this has, on the whole, been circumvented by accepting that a stage (or two) of the contract be only partially completed (a refinement of part of the abstract specification, for example) and by the introduction of a flexibility weighting. From the contractee's point of view, supervising what appears to be a large number of student projects requires a deal of effort and concentration. The contractee will have to become familiar, almost instantly, with a variety of systems that students wish to model. During workshops the contractee may feel rather like a chess master playing many games simultaneously! With most classes it is better that students work in groups - only if the class is very small (say, less than ten students) should individual contracts be attempted. Even with seven or eight contracts it has been found that, without a second "consultant" on hand, four hours per week is barely enough time to see each group.

As we have stated, learning contracts are not new. Since their use on mathematics and modelling courses has been limited, it was felt worthwhile to describe here the success at Sheffield in employing contracts as a means of learning mathematical concepts and techniques and of acquiring wider, transferable skills. Whilst the application of mathematical modelling to software engineering has been emphasised, it should be noted that learning contracts could equally well be applied to any modelling application or area.

References

1. McLone R, "The Training of Mathematicians", SSRC Report, 1973

2. Council for Industry and Higher Education, "Towards a Partnership: Higher Education - Government - Industry", 1987

3. National Advisory Body, "Transferable Personal Skills in Employment - The Contribution of Higher Education", 1986.

4. MSc Engineering Information Technology, Definitive Course Document, School of Engineering Information Technology, Sheffield Hallam University, 1989.

5. Spivey J M, "The Z Notation: A Reference Manual", Prentice Hall International Series in Computer Science; 1989, 1992 (2nd Ed), ISBN: 0-13-978529-9

6. Norcliffe A and Slater G, "Mathematics of Software Construction", Ellis Horwood Series in Mathematics and its Applications, 1991, ISBN: 0-13-563370-2 (Library Edn) ISBN: 0-13-563388-5 (Student Pbk. Edn)

7. Diller A, "Z: An Introduction To Formal Methods", John Wiley and Sons, 1990, ISBN: 0-471-92489-X

8. Wordsworth J B, "Software Development with Z", Addison-Wesley, 1992, ISBN: 0-201-62757-4

APPENDIX A.5: A Systematic Approach to Writing Simple Z Specifications

[And95]

l)CNG Iflt-M

hematic Approach to Writing Simple Z Specifications

SoRRY

Norcliffe and Simon Andrews
ematics Centre
>d of Engineering Information Technology
zield Hallam University
Campus
neld SI 1WB

S c.fZ K 'i

—act

A systematic method of arriving at the state schema of a Z specification is presented. The OPERATOR method, as it is known, is based on many years' experience of teaching Z and has been designed to ease students through the problems they face identifying system state variables and modelling them in Z. The method draws on well-proven ideas of structured methods but is essentially free standing. The method, in its simplest form, has been tested on students and we report on our experience of using the method in the classroom. We discuss the good points and limitations of the method and show how the method may be enhanced by means of an accompanying diagramming notation and how it may be extended to address system complexify. Future related research areas are identified.

Introduction

ougk most university computing courses now include a study of the Z specification age, the teaching of Z is still not without its problems. Students not only find difficulties ' tg to grips with the notation and underpinning mathematics, but experience enormous ■lems when they first come to use the notation to construct a system specification from requirements. Students find it hard coping with abstraction and identifying the particular Lbles that make up the system state. The act of getting started seems to be the problem once through this bottleneck (Norcliffe, 1993), and having produced the state schema, -ents find it much easier to specify the associated operation and error-handling schemas, usually go on to complete the specification with few further problems. They may not sdiately specify the operations correctly, but at least seem more comfortable with this part *-e process - probably because it is more mechanical and there is less need of abstraction.

_JL the specific aim of easing students through this abstraction bottleneck, a simple method been developed at Sheffield Hallam University to enable students to create the system ~e schema in a systematic way. The OPERATOR method, as it is known, was developed z'ntly by the authors and is based on many years' experience of teaching Z. The method ~'vs on well-proven ideas of structured methods (Hares, 1990; Yourdon, 1989), but does require a frill-blown structured approach to be carried out first. The method is essentially standing and is therefore different to the formal and structured methods integration roaches developed recently (Semmens & Allen, 1991; Randell, 1991; Polacketal, 1993). e method, in its simplest form has been tried out on second year degree students at effield, and is presently the subject of ongoing research.

's paper we describe in detail how the OPERATOR method works. Its use in developing schemas is illustrated in the next two sections where a simple security system and 'ng system are considered. In Section 4 we comment briefly on our experience of using method with our second year students and list what we feel are the good points of the od and its limitations. We also look at ways in which the limitations might be removed in Section 5 we show how the OPERATOR approach can be enhanced by using simple ams. In Section 6 we consider ways of addressing system complexity and show how the od, and its graphical front end, can be extended to tackle the specification of complex ems. Section 7 is a conclusion.

The OPERATOR method: a simple example

see how the OPERATOR method works we use it to produce the state schema needed in specification of a simple security system. Assume the system we are to specify has the owing user requirements.

> The system is to monitor the whereabouts of staff in an organisation. The organisation is located in its own building and, as staff check themselves in and out of the building, the system notes whether they are in or out as appropriate. The system can be queried at any time to see who is in or out, and must cope with staff joining and leaving the organistion.

e word OPERATOR is an acronym with the letters standing for Objects, Properties, tities, Relationships, Assemble, Trim, Other and Repeat. Step 1 of the method therefore gins by identifying the objects that make up the system. In our example obvious candidates r objects are the staff who work in the building. Whilst it is not imperative that all objects identified at this stage - indeed, the later identification of objects is an integral part of the ethod - it is worth noting that there are no other obvious objects making up the system that d concern us. It is worth noting, too, that we need not be overly strict about what nstitutes an object other than that objects should be nouns and have some concrete 'stance (Sully, 1993).

ep 2 of the method requires us to identify the **properties** of these objects. At this stage it is)ortant to note that we are looking only for simple has/have properties. Other relationships established during step 4 of the method. From the requirements of our system it is clear t staff have whereabouts, and it is this property that the system must monitor. Staff in the rganisation have no other properties of significance and thus we can proceed to step 3 and entify the entities making up the system.

ıe entities of the OPERATOR method are the nouns identified in steps 1 and 2. The entities e thus the staff and their whereabouts. As part of this third step we must also describe the tities in terms of the Z notation. Basic types are therefore needed and we parachute in the pe set *STAFF_ID* and introduce the enumerated type *IN_OUT* containing the elements *in* d *out*. The system entities *staff* and *whereabouts* are thus declared as follows:

$$\begin{aligned} staff &: \mathbb{P}\,STAFF\_ID \\ whereabouts &: \mathbb{P}\,IN\_OUT \end{aligned}$$

stem entities *staff* and *whereabouts* are sets of *STAFF_ID* and *IN_OUT* values tively, thus explaining the use of the powerset symbol in the declarations. The variables and *whereabouts* are possible state variables; additional state variables are identified via of the method, where relationships between system entities are established.

tionships between entities are identified in a systematic way using the concept of the /entity matrix shown below. At this stage the aim is to identify binary relations only. e complicated relationships are introduced via the data invariant of the state schema once te variables have been identified.

	staff	*whereabouts*
staff	-	*location_of*
whereabouts	*located*	-

ch cell of the matrix are put names of relevant binary relations between the pairs of es involved. The assumption is that the entity in the row of the matrix is the domain of elation, and the entity in the column is the range. Where system entities are not sets but e elements, they should be regarded as singleton sets if binary relations involving them are ed. In practice this seldom happens. Thus, *location_of* is a binary relation between staff their whereabouts and, since at any time staff have unique locations, then the binary tion is actually a partial function with the following declaration:

$$location\_of \quad : \quad STAFF\_ID \nrightarrow IN\_OUT$$

binary relation *located* is not a function as several staff may be in or out of the building at one time. Its declaration is this:

$$located \quad : \quad IN\_OUT \leftrightarrow STAFF\_ID$$

should note that the remaining cells of the matrix are empty because no relevant relations betwen the entities concerned.

5 of the method is to assemble the list of candidate state variables. This list contains the em entities together with the binary relations identified. The assembled list of variables d their declarations is thus as follows:

$$\begin{array}{rcl} staff & : & \mathbb{P}\, STAFF\_ID \\ whereabouts & : & \mathbb{P}\, IN\_OUT \\ location\_of & : & STAFF\_ID \nrightarrow IN\_OUT \\ located & : & IN\_OUT \leftrightarrow STAFF\_ID \end{array}$$

is more than likely that this list is longer than it need be and, so, in step 6 of the method we n it down. The trimming is achieved systematically by getting rid of redundant ormation. We can usefully note that *staff* is the same as dom *location_of*. Thus, if we h we need not include *staff* in the state schema provided we include the partial function *ation_of*. Similarly, we need not include the set *whereabouts* because this is the same as *location_of*. Finally, we need not include *located* because this is just the inverse of *ation_of*.

ry, then, all the information we might need is contained within the *location_of* function. 'er, it may be sensible to include *staff* in the abstract specification even though the ation is redundant, so that a direct record of the users of the building is ready to hand ecification purposes. The level of redudant information is really a matter of taste. y it should not be great, but at the same time it is important to ensure that specifications ıdable and easily understood (Gravell, 1991; Spivey, 1992). The trimmed list of state bles is thus:

$$\begin{aligned} staff &: \mathbb{P}\, STAFF\_ID \\ location\_of &: STAFF\_ID \rightarrow\!\!\!\!+ IN\_OUT \end{aligned}$$

remaining 2 steps of the method require us to check whether there are other objects of , and to repeat the process with them included. Fortunately there are no other objects ıerefore no need to repeat the process. Repeating the process is in principle not difficult. should be taken to check for additional relationships between new and existing entities g the repeat step 4.

culmination of the OPERATOR method is thus the listing of state variables given above. state variables and the properties which they possess can now be set down in the system e schema:

System

$staff : \mathbb{P}\, STAFF\_ID$
$location\_of : STAFF\_ID \rightarrow\!\!\!\!+ IN\_OUT$

$staff = \mathrm{dom}\ location\_of$

e OPERATOR method will not determine the data invariant. However, having identified state variables, the data invariant can be determined from knowledge about the system and Z constructs used to model the system variables.

om hereon, the rest of the system specification can be established. This will include ecifying state changing operations such as checking in and checking out of the building by f, adding new staff members and removing staff from the system when, for example, they 'e the organisation. Querying operations, which do not change the state, can similarly be ecified, and might include such operations as querying the system to see who is in or out of e building.

nother simple example

emonstrate the applicability of the OPERATOR method we consider, albeit briefly this , another example - that of a simple banking system. The requirements of the system, we ıe, are the following.

> The balances and overdraft limits of accounts at a bank are to be monitored by the system. Account holders can make deposits and withdrawals and, if they have sufficient funds, can change their overdraft limits. As well as furnishing information on balances and overdraft limits, the system should cope with opening and closing accounts.

lication of the OPERATOR method, with brief annotations, is as follows.

jects: *accounts*, *holders*

perties: *accounts* have *balances*
accounts have *od_limits*

tities:

holders : $\mathbb{P}$ *HOLDER_ID*
accounts : $\mathbb{P}$ *ACC_NO*
balances : $\mathbb{P}\mathbb{Z}$
od_limits : $\mathbb{P}\mathbb{Z}$

e we should note that the type set $\mathbb{Z}$, representing the integers, is being used to model the nces and overdraft limits (in pence) of individual accounts. Other types used have their ious meanings

elationships:

	holders	*accounts*	*balances*	*od_limits*
holders	-	*account_of*	-	-
accounts	*holder_of*	-	*balance_of*	*od_limit_of*
balances	-	-	-	-
od_limits	-	-	-	-

semble:

holders : $\mathbb{P}$*HOLDER_ID*
accounts : $\mathbb{P}$*ACC_NO*
balances : $\mathbb{P}\mathbb{Z}$
od_limits : $\mathbb{P}\mathbb{Z}$
account_of : *HOLDER_ID* $\nrightarrow$ *ACC_NO*
holder_of : *ACC_NO* $\leftrightarrow$ *HOLDER_ID*
balance_of : *ACC_NO* $\nrightarrow \mathbb{Z}$
od_limit_of : *AC_NO* $\nrightarrow \mathbb{Z}$

ote that the concept of joint accounts is being modelled by declaring *holder_of* to be a binary elation and not a partial function. By declaring *account_of* to be a partial functio n, the ssumption is that holders can only hold one account.

:

holders	:	$\mathbb{P}$*HOLDER_ID*
account_of	:	$HOLDER\_ID \rightarrowtail\!\!\!\!\!\!\!+ ACC\_NO$
balance_of	:	$ACC\_NO \rightarrowtail\!\!\!\!\!\!\!+ \mathbb{Z}$
od_limit_of	:	$AC\_NO \rightarrowtail\!\!\!\!\!\!\!+ \mathbb{Z}$

ıming the list we have noted that *accounts* is dom *balance_of*, that *balances* is ran ;e_of, that *od_limits* is ran *od_limit_of*, and that *holder_of* is the inverse of the function ıt_of.

r: There are no other objects of note.

eat: This step is unnecessary.

state schema, with its appropriate data invariant is as follows:

```
──── Bank ──────────────────────────────
│
│ holders : ℙHOLDER_ID
│ account_of : HOLDER_ID ⤔ ACC_NO
│ balance_of : ACC_NO ⇸ ℤ
│ od_limit_of : ACC_NO ⇸ ℤ
│
├──────────
│
│ holders = dom account_of
│ ran account_of = dom balance_of
│ dom balance_of = dom od_limit_of
│ ∀x : dom balance_of •
│ (balance_of(x) ≥ od_limit_of(x)
│ ∧ od_limit_of(x) ≤ 0)
│
└────────────────────────────────────────
```

te that the data invariant is reflecting the operating assumptions of a normal bank - namely t all accounts have balances and overdraft limits, that overdraft limits should be not ›eded, and that overdrafts represent negative amount of cash. Once again, to complete the ification, operations that change the state of the system, and those which only query the e, would now be specified.

Using the method: its good points and its limitations

e method was first tested on second year Computing Mathematics degree students at effield Hallam University. Students had already been exposed to discrete mathematics and e Z notation, and were familiar with reading Z specifications. They had, for example, died the video-based Z Readers course produced at Sheffield (Cooper et al, 1992) and ew how specifications were structured. They had not, however, had any experience of 'ting specifications and the OPERATOR method was the first systematic approach they had ed to develop Z specifications.

'ng in small groups students had to specify a simple library system. An extract from the user requirements document is as follows:

> In order to monitor who the users of the library are, which copies of books they have on loan, and which copies are available for borrowing, a simple computer-based system is to be developed. Any copy of a book that has been borrowed will have a return date stamped inside it and this will be noted by the system. The system must also log the acquisition of new copies of books and note their removal, and should enable new users to join the library and existing users to leave.

marks for the complete (non-robust) specification were 50, of which 10 were available for of the OPERATOR method to determine the list of state variables and their declarations. average mark for use of the OPERATOR method was 7.41 with a standard deviation of . The marks ranged from 4 to 9 and there were 17 groups of students. Most succeeded ·ing the method well and produced a variety of consistent specifications. Most lists of variables were variations on the following:

$users$	:	$\mathbb{P}USER\_ID$
$copies$	:	$\mathbb{P}COPY\_ID$
$books$	:	$\mathbb{P}BOOK$
$borrower\_of$	:	$COPY\_ID \nrightarrow USER\_ID$
$book\_of$	:	$COPY\_ID \nrightarrow BOOK$
$status\_of$	:	$COPY\_ID \nrightarrow STATUS$
$duedate\_of$	:	$COPY\_ID \nrightarrow DATE$

eral groups had been harsher with their trimming than others and had removed *copies* and *ks*. Others had introduced the concept of library cards and additional information about ks such as their titles and authors. A common omission was the *status_of* function which 'cates whether a book is available for borrowing or not. Since its inclusion in the ification is not essential, the omission was not penalised.

summary students found the method easy to understand and simple to use. The method had ı demonstrated using the examples considered in Sectons 2 and 3, and students were able apply the ideas readily to develop the simple library system. Many of the specifications ed out similar as a result of applying the method, although there had been minimal copying ideas by groups. Whether this high level of reproducibility is a good feature of the method debatable. The approach certainly steers the specifier towards the use of functions and tions when perhaps simpler structures might have been used. The security system, for mple, is easily developed in terms of just sets (Norcliffe & Slater, 1991; Cooper et al, 92). Students commented that they found the method enabled them to construct ecifications in a systematic way. In general they found this helpful and were able to have sible discussions about the system based around the approach being adopted.

though the comments of the students were positive in the main, the method does have its itations. The approach, though systematic, is still very abstract. It is interesting to note at some students were drawing informal diagrams to help them apply the method. Given at the success of structured methods such as SSADM, Jackson, and Yourdon seem to hinge the use of accompanying diagrams, the authors deemed it necessary that the OPERTOR

d should also have a diagramming notation. In the next section we therefore show
ly how the method can be enhanced through the use of diagrams.

ll as being abstract, the approach as outlined so far does not really address system
lexity. In discussions the students commented that they felt the method could soon
e unworkable if the number of entities became large. Drawing up a large entity/entity
would be difficult, for example, and ensuring that the data invariant of a large state
a was correct would also not be easy. In Section 6 we therefore show how the
ATOR method, and its diagramming notation, can be extended to address system
lexity and to embrace structural considerations such as partitioning a system into several
stems.

nhancing the method: a graphical front end

graphical notation described in this section has been developed to accompany the method
o facilitate the O, P and E stages. Its use is therefore designed to help identify the objects
entities that make up the system being specified. The notation is as follows:

ıe system at the top level is represented by an appropriate descriptor written inside a
ectangular box as shown:

$$\boxed{\textit{System}}$$

convention is that the first letter of the discriptor is an upper case letter. If we were
eloping a banking system we may well expect to see *Bank* written inside the box nstead of
em.

Objects and other system entities, related by the has/have property, are also represented by
names written inside rectangular boxes:

$$\boxed{\textit{Staff}}$$

convention with system entities (objects are also entities) is that their names are written in
er case throughout.

Each of the boxes representing an entity has the set, to which the entity belongs, written
alongside in Z, eg:

$$\mathbb{P}\ \mathit{STAFF\_ID}$$

e convention here is that types and other sets used are written in upper case letters
oughout, and are not contained inside boxes.

The hierarchical relationships between the above are represented by arrows of appropriate
kinds:

$\longrightarrow$ links the system at the top level to objects of which it is comprised.

$\longrightarrow$ links objects to entities, and entities to their associated entities as appropriate. The arrow characterises the has/have property.

$\dashrightarrow$ links entities (including those identified as objects) to the sets in Z to which they belong.

ow how the notation works, let us draw the diagrams that represent the security, ıg and library systems considered earlier. Figure 1 shows the simple security system.

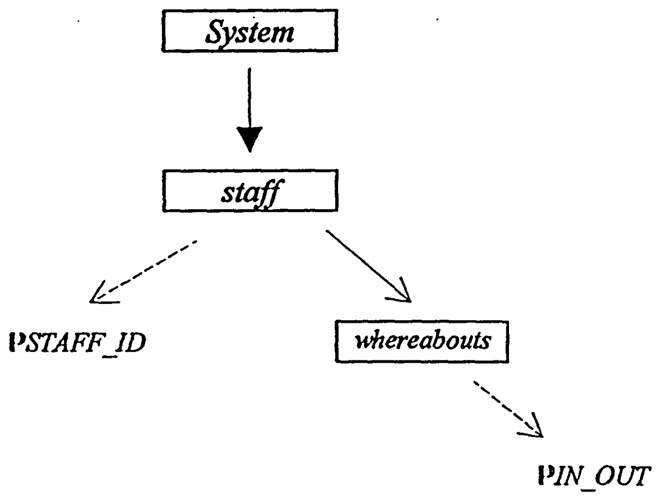


Figure 1 - Graphical representation of the simple security system

e diagram tells us that the system state at the highest level is called *System*. The objects in system are *staff* who have *whereabouts*. The system entities are therefore *staff* and *reabouts*, and these are possible state variables. The variable *staff* is a member of the nstruted type set P*STAFF_ID*, and whereabouts belong to P*IN_OUT*.

ıre 2 shows the banking system and Figure 3 gives one interpretation of the simple library ;em.

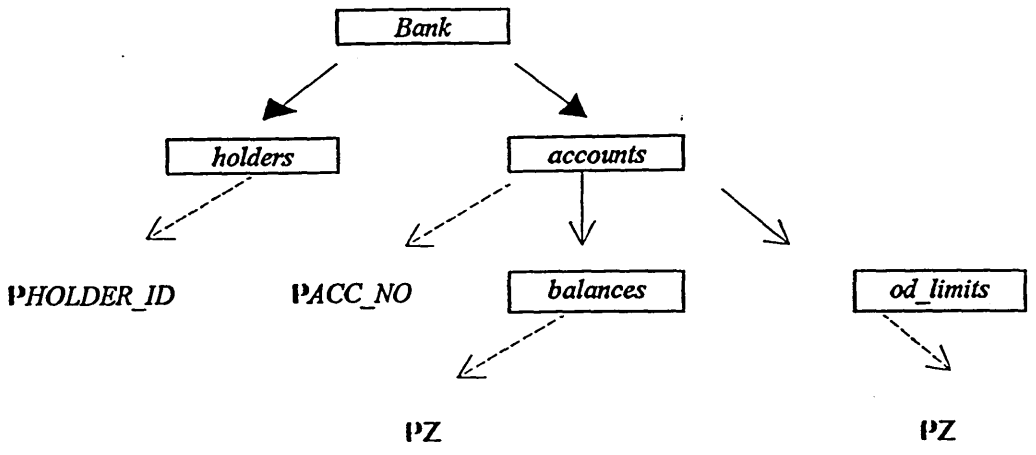


Figure 2: Graphical representation of the banking system

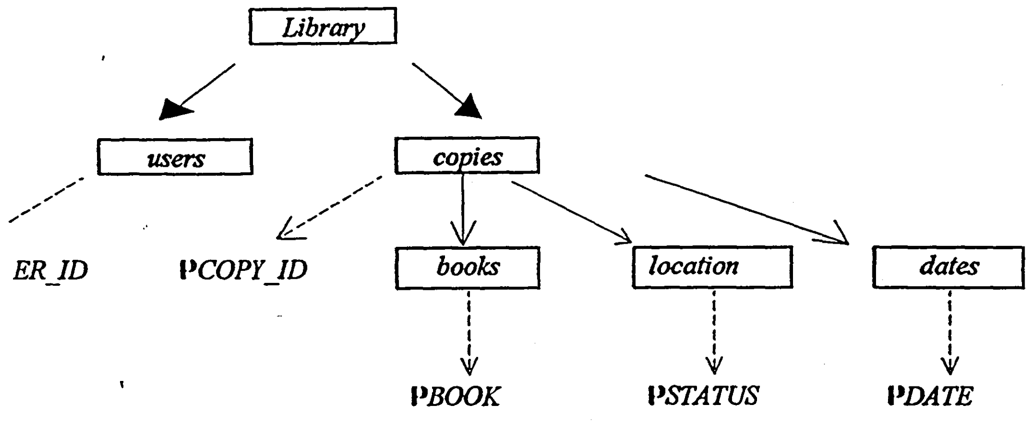


Figure 3: Graphical representation of the library system

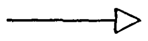
pefully the diagrams speak for themselves. It should be noted that different diagrams may
ll lead to the same Z specification. In Figure 2, for example, it is assumed that *holders* and
ounts are both objects. There is nothing wrong with a diagram that regards just *holders* as
jects and *accounts* as associated entities - in the sense that account holders have bank
›unts. Since *holders*, *accounts*, *balances* and *od_limits* emerge as the system entities
er way round, the odds are that the resulting Z specifications of the state will be the same.
e prime purpose of the diagramming notation is to assist the specifier to identify system
tities, and this we feel it does. The strength of the notation is that it is graphical and
erarchical, and readily enables a picture of the system state to be created showing explicitly
e entities that are part of it.

ddressing complexity

ṣ a method can be used to develop large systems, and therefore cope with complexity, it ly no method at all. In this section we indicate briefly how the method and the ated graphical notation can be extended to cope with the specification of complex ıs. The ideas in this section are relatively new and are the subject of ongoing rsearch. y issues have yet to be resolved and the applicability and effectiveness of what is being ested have yet to be tested and evaluated.

address complexity by partitioning a system into appropriate subsystems and applying the ATOR method to each subsystem in a "divide and conquer" fashion. To do this the amming notation requires a new kind of rectangular box and a new kind of arrow. The kind of box is one containing a subsystem name. Thus, in the case of the library system, were felt that a partitioning of the library into three subsystems, namely *Users*, *Copies* and ṣ, was needed, then a typical subsystem box would be the following:

new arrow that is needed is the following one:



ch links a system to its subsystems.

see how the ideas can be applied, let us revisit the library and think of it not as a monolithic em, with no real structure, but comprising the three subsystems proposed above. This of the library is illustrated diagrammatically in Figure 4, where the extended subsystem tion is used.

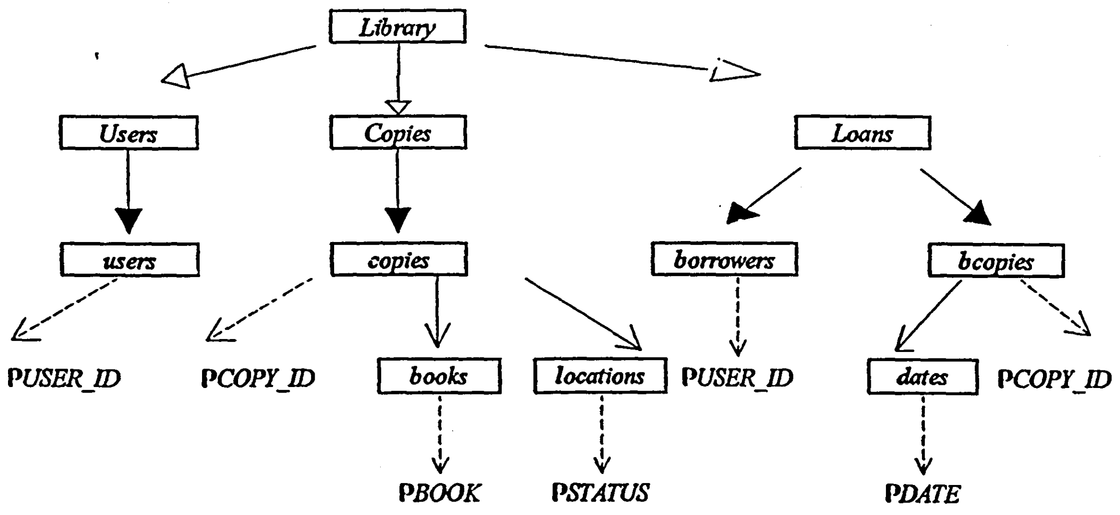


Figure 4: Partitioned view of the library system

PERATOR method can now be used to develop substate schemas to specify the states *Users*, *Copies* and *Loans* subsystems. The state schema of the Library is then the which includes these three substate schemas. Application of the OPERATOR d, as described earlier, leads to the following *Users*, *Copies* and *Loans* substate is. Their derivation is straightforward and they are presented without explanation. In *ms* subsystem note that new variables ***borrowers*** and ***bcopies*** (borrowed copies) have introduced.

Users

$users : \mathbb{P}\,USER\_ID$

Copies

$copies : \mathbb{P}\,COPY\_ID$
$books : \mathbb{P}\,BOOK$
$book\_of : COPY\_ID \nrightarrow BOOK$
$status\_of : COPY\_ID \nrightarrow STATUS$

$copies = \mathrm{dom}\; book\_of$
$\mathrm{dom}\; book\_of = \mathrm{dom}\; status\_of$
$books = \mathrm{ran}\; book\_of$

Loans

$borrowers : \mathbb{P}\,USER\_ID$
$bcopies : \mathbb{P}\,COPY\_ID$
$borrower\_of : COPY\_ID \nrightarrow USER\_ID$
$duedate\_of : COPY\_ID \nrightarrow DATE$

$borrowers = \mathrm{ran}\; borrower\_of$
$bcopies = \mathrm{dom}\; borrower\_of$
$\mathrm{dom}\; borrower\_of = \mathrm{dom}\; duedate\_of$

ese three schemas can now be included into one *Library* schema to create the state schema the library system. The data invariant serves to relate all the variables involved defining, in icular, the status of copies of books that have been borrowed, and those which should be ailable for borrowing:

Library

Users
Copies
Loans

$borrowers \subseteq users$
$bcopies \subseteq copies$
$\forall c : bcopies \bullet status\_of(c) = borrowed$
$\forall c : copies \bullet c \notin bcopies \Rightarrow status\_of(c) = available$

ntrast, and for comparison, the state schema of the monolithic unpartitioned library m, again developed using the OPERATOR method, is as follows:

Library

$users : \mathbb{P}\,USER\_ID$
$copies : \mathbb{P}\,COPY\_ID$
$books : \mathbb{P}\,BOOK$
$borrower\_of : COPY\_ID \mathrel{+\!\!\!\!\rightarrow} USER\_ID$
$book\_of : COPY\_ID \mathrel{+\!\!\!\!\rightarrow} BOOK$
$status\_of : COPY\_ID \mathrel{+\!\!\!\!\rightarrow} STATUS$
$duedate\_of : COPY\_ID \mathrel{+\!\!\!\!\rightarrow} DATE$

$\mathrm{ran}\ borrower\_of \subseteq users$
$\mathrm{dom}\ borrower\_of \subseteq copies$
$\mathrm{dom}\ book\_of = copies$
$\mathrm{ran}\ book\_of = books$
$\mathrm{dom}\ book\_of = \mathrm{dom}\ status\_of$
$\mathrm{dom}\ duedate\_of = \mathrm{dom}\ borrower\_of$
$\forall c : \mathrm{dom}\ duedate\_of \bullet status\_of(c) = borrowed$
$\forall c : \mathrm{dom}\ book\_of \bullet (c \notin \mathrm{dom} duedate\_of \Rightarrow status\_of(c) = available)$

fore leaving this section, it is worth pointing out that simple diagrams, akin to data flow grams, can be drawn to help guide the construction of system operation schemas. Below in gure 5 we sketch out a possible representation of the BorrowCopy operation in the titioned library system, whereby a user borrows a copy of a book.

gure 5:

e diagram indicates the inputs to the BorrowCopy operation along with their associated es. It shows that state variables in Copies and Loans will be changed (if the operation is ssful) and that the information held in Users and Copies will be needed to check whether the operation will succeed.

onclusions

paper we have presented a method of arriving at the state schema of a Z specification in ematic way. The method, in its simplest form, was tested on students who used it in the lopment of a simple library system. Students found the method helpful, but felt that the od would soon become difficult to use with increasing system complexity. In the paper ve therefore shown how the method may be extended to address system complexity and it may be enhanced by means of an accompanying diagramming notation.

its graphical front end, and its system partitioning capability, we feel that the ATOR method has potentially much to offer the Z user. The method is easily stood and relatively easy to use. The diagramming notation encourages the specifier to igate key structural issues at an early state and brings in formal data modelling ideas in a l way. Once the developer is happy that structural issues have been addressed, the R, A, and R stages of the method can be used to furnish the state schema in a systematic and ent way.

e have said, the OPERATOR method is the subject of ongoing research. The approach rently being refined and extended to include the specification of system operations along ies outlined above in Section 6. The potential for embedding the method in a simple use-dly CASE tool is also being investigated.

ences

ooper D, Mardell J, Meehan A, Norcliffe A and Valentine S, 1992, A Z Readers Course a video-based training course for programmers in industry, PAVIC Publications, Sheffield.

Gravell A M, 1991, What is a good formal specification?, in Nicholls J E (ed): Proceedings of the Fifth Annual Z User Meeting, pages 137-150, Springer-Verlag, London.

Hares J S, 1990, SSADM for the advanced Practitioner, John Wiley & Sons, Chichester.

Norcliffe A, 1993, Computer Aided Modelling, in Yardley P (ed): Proceedings of the Eighteenth Undergraduate Mathematics Teaching Conference, pages 4-24, Shell Centre Publications, Nottingham.

Norcliffe A & Slater G L, 1991, Mathematics of Software Construction, Ellis Horwood, Chichester.

Polack F, Whiston M & Mander K, 1993, The SAZ project: Integrating SSADM and Z, in Woodcock J C P & Larson P G (eds): FME '93 - Industrial-Strength Formal Methods, Lecture Notes in Computer Science, pages 541-557, Springer-Verlag, London.

Randell G P, 1991, Data flow diagrams and Z, in Nicholls J E (ed): Proceedings of the Fifth Annual Z User Meeting, pages 216-227, Springer-Verlag, London.

Semmens L T & Allen P M, 1991, Using Yourdon and Z: An approach to formal specification, in Nicholls J E (ed): Proceedings of the Fifth Annual Z User Meeting, pages 228-253, Springer-Verlag, London.

Spivey J M, 1992, The Z Notation: A Reference Manual. Series in Computer Science, Prentice-Hall International, 2nd edition, Hemel Hempstead.

Sully P, 1993, Modelling the World with Objects, Prentice-Hall International, Hemel Hempstead.

1 Yourdon E, 1989, Modern Structured Analysis, Prentice-Hall International, Englewood Cliffs.

ZAPPA - A Tutorial Introduction

In this tutorial you will use ZAPPA, a prototype CASE tool for the animation of formal specifications written in Z, to animate part of a simple banking system (you should be fairly familiar with the bank spec. by now).

Along with this guide you will need:

- a copy of the bank spec. which you should continuously refer to during the tutorial,

- A ZAPPA function listing - basically the ZAPPA version of the part of the Z mathematical toolkit required for the bank spec.

You will, of course, also require a copy of ZAPPA.

During the tutorial you will animate the state schema *Account*, the initial state schema *InitAccount*, the operation schemas *Open*, *Close*, *Deposit* and *Withdraw* and the robust operation schemas, *ROpen* and *RDeposit*.

Do not hesitate to ask for assistance!

ZAPPA should be up and running on your machine and you should be looking at the amber/brown ZAPPA ANIMATOR screen.

As the name implies the ANIMATOR is used to create (and delete) the various objects and structures that make up a Z specification. By "pressing" the appropriate grey buttons on the screen it is possible to create, for instance, operation schemas, state variables, input variables etc.

It is also possible to move to the other two ZAPPA screens from the ANIMATOR screen, the other two being the OVERVIEW and the ANIMATION.

"Press" the OVERVIEW button by moving the arrow pointer with the mouse onto the button and clicking the left hand mouse button **once**.

[NOTE: In the rest of this tutorial "click on X" means point to X using the mouse and click the left hand mouse button once.]

The OVERVIEW screen (black text on white background) should now be displayed.

The OVERVIEW, as the name suggests, gives an overall view of the objects and structures of an animated specification.

You can see the various categories o o ject, or examp e **OPSchemas** (operation schemas), **ParaTypes** (parachuted basic types) and **ErrSchemas** (error schemas).

At the moment the OVERVIEW should be "blank", ie no objects or structures have yet been created.

[NOTE: Ignore **Global**, **Image** **KWindow** **zeros** and **Temp.**]

Return to the ANIMATOR by clicking on the small grey square with the downward pointing arrow ▼ in the top right hand corner of the screen.

The third ZAPPA screen is the animation screen [illegible] (green background).

Click on the ANIMATION button to display the screen.

The ANIMATION screen is used to "run" the animated specification as robust operation schemas are constructed.

Again, we should be starting with (apart from the four utility buttons in the lower left hand corner) a blank ANIMATION screen.

The first task in animating the bank spec is to parachute *ACC_No* into the animated spec.

Click on the **animator** button lto return to the ANIMATOR screen.

Click on **Parachute Type** (centre, top button).

You will be prompted to enter the name of the type, so enter **ACC_NO** and press the return key (or click on **OK**).

[NOTE: If you make a typing mistake when entering information in ZAPPA you can delete characters to the left of the flashing input cursor by using the backspace key 🡐.

If you create something erroneously or by mistake you can use the appropriate **Delete** button to delete it.

Several of the pop-up menus on ZAPPA also have a *CANCEL*.]

ZAPPA will tell you that *ACC_NO* has been parachuted into the spec. Press return again (or click on **OK** in the message box).

Check that *ACC_NO* has been parachuted by looking at the OVERVIEW. *ACC_NO* should be connected to **ParaTypes**.

3 Creating the State ⌞chema, ccoun

Unlike other schemas, in ZAPPA a state schema is created in three stages:

- the schema box
- the state variables (signature)
- the data invariant (predicate)

(a) Creating the schema box

Click on Create **State Schema Box** and enter the name of the state schema - *Account*

(b) Creating the state variables

55

Account has two state variables, *balance* and *od-limit*. Their declarations are:

$$\begin{aligned} &balance\colon ACC\_NO \longrightarrow\!\!\!\!\!\!\mapsto \mathbf{Z} \qquad \text{and} \\ &od\_limit\colon ACC\_NO \longrightarrow\!\!\!\!\!\!\mapsto \mathbf{Z} \end{aligned}$$

Click on **Create State Variable** and enter *balance*. You will thenbe prompted to select the sort of variable. *Balance* is a function so click on **Function** in the pop-up menu.

You are then prompted to select the structure of the function. It is a function from one basic type to another so click on **A —⇸ B** in the menu.

(A). For *balance* they are of type *ACC_NO* so click on **ACC_NO** in the menu.

And for **B**, the range element type, select **Z**, the set of all integers.

ZAPPA will then tell you that *balance* has been created, displaying its declaration.

Now create *od_limit.*

[NOTE: Use OVERVIEW regularly to check on the progress of the animation.]

(c) Creating the data invariant

In the ANIMATOR, click on **Create Data Invariant.** ZAPPA then instructs you on how to proceed. Press return - a small window called KTOOLS appears. Click and hold down the left mouse button on **Function** in the KTOOLS window and then "drag" the arrow pointer down the pop-up menu that appears, to highlight **Edit.** Release the mouse button and an **Edit Function** menu appears. Click on **DICheck** to highlight it and then click on **OK.**

You shou__ t_en __ t_ en to e unc on e simple template for the data invariant:

```
{
If NULL
        Then zmessage ("data invariants ok")
        Else zmessage ("data invariant error");
zend();
};
```

We now need to translate the data invariants of *Account* from Z to ZAPPA's mathematical toolkit equivalents, and enter them where the **NULL** is in the template.

Click on the template, just after the **NULL** to position the flashing text editor cursor. Erase **NULL** by pressing the backspace key four times.

We now need to translate the data invariants in *Account* into ZAPPA.

The first invariant is straightforward, it is the one that ensures that every account has a balance and an overdraft limit:

dom *balance* = dom *od_limit*

In ZAPPA all Z toolkit functions begin with the letter "Z" and, as far as possible, conform to a direct or natural language translation. For example:

dom translates simply **zdom** in ZAPPA.

In-fix functions and relations translate to post-fix functions in ZAPPA, hence

$x = y$ translates as
zequal? (x,y)

the question mark indicating that the function returns TRUE or FALSE.

So, using nesting of functions

dom *balance* = dom *od_limit*

translates into

zequal? (**zdom**(balance), **zdom**(od_limit))

in ZAPPA.

Data invariants are entered into thre **DICheck** template in the format

```
If      (invariant 1    And
         invariant 2    And
              :
         invariantn)

Then ...
```

So open brackets by typing ((your cursor should already be in the correct position after erasing **NULL**) and enter the first invariant as translated above, followed by **And** and a return. Position the cursor ready for the second invariant directly under the **z** in **zequal?** using the space-bar. (This is not essential, it just looks better!)

The second data invariant in *Account* is

$$\forall x : \text{dom}\, balance \bullet balance\ x \geq od\_limit\ x$$

Skipping this for a moment, the third invariant is

$$\forall x : \text{dom}\, od\_limit \bullet od\_limit\ x \leq 0$$

To animate this we need to use the ZAPPA function **zfor_all_01.** one of (eventually!) a library of quantified functions in ZAPPA.

Refer to your ZAPPA function list for a description of **zfor_all_01.**

Noting that the format for **zfor_all_01** is

zfor_all_01 (dum,set,val1,rel,val2)

the translation of the third data invariant is

zfor_all_01 (x, **zdom** (od_limit), od_limit,
< = , zeroZ)

zero is a part o , a zer .
zeroZ rather than the value 0 because ZAPPA needs to know what sort and type a variable or object is.

Enter the translated data invariant, and, using **zfor_all_01** again, translate and enter the second data invariant - don't forget the **And** and to close brackets after the last invariant.

When you have finished click on the small grey box at the top left hand corner of the Function Editor and click on **Close** in the pop-up menu. You are then asked if you wish to save the function. Click on **YES**.

[NOTE: The data invariant, initial state, operation, error and robust operation schemas, once created can be edited at any time by clicking on **KTOOLS** in the ANIMATOR window and choosing **Function** then **Edit** followed by the appropriate name.]

Notice that the two universally quantified data invariants used a dummy variable called *x* as in

$$\forall x : \text{dom}\, balance$$

which is shorthand in Z for

$$\forall x : ACC\_NO \mid x \in \text{dom}\, balance$$

In ZAPPA we need to create a dummy variable for **zfor_all_01** to use.

Click on **Create Local/Dummy Variable** in _e and ter x as the variable name.

Next, select **tuple** and then A as the variable structure and **ACC_NO** as the type. ZAPPA will then ask if you wish to give the variable a value. Because we are using x as a dummy variable we do not wish to give it a value so click on **NO**.

4 Creating the Initial State Schema

Click on **Create Initial State** and enter the schema name **InitAccount.**

Follow the same editing procedure as before to enter the function editor to edit the simple initial state schema template.

58

We now have to enter the predicate of *InitAccount*, which, in Z, is

$$balance = \emptyset$$
$$od\_limit = \emptyset$$

Position the cursor in front of the first curly bracket and press **return.**

In ZAPPA we have a special function that empties a set, **zmake_empty.**

Enter the two predicates, noting that each one must end with a semi-colon, eg

Save the function as before.

At this point look at the ANIMATION screen by clicking on ANIMATION.

Note that two buttons have appeared, **InitAccount** and **DICheck.**

Click on **InitAccount** to initialise the state.

[NOTE: The state may be initialised at any time - this is especially useful if, through a mistake in the specification, the data invariant is violated. You can always start afresh.]

Click on **DICheck** to make sure that the initial state satisfies the data invariant. Be patient - **DICheck** can be slow!

We can look at state variables at any time using the **show var** button.

Click on **show var** and check that *balance* is empty. Do the same for *od_limit*. Note that after viewing a variable you must click on **END** in the bottom right hand corner of the screen.

Return to the ANIMATOR.

5 Creating *ROpen*

(a) Creating *Open*

Firstly we need to create two input variables for open

New? : ACC_NO and
Odl? : Z

Click on **Create Input Variable** and enter **new?**. Select **tuple, A** and **ACC_NO.** Because we are creating an input variable ZAPPA asks for a prompt for the variable. Enter **Enter new account number.**

Follow a similar procedure to create **odl?**.

Now we are ready to create **Open.**

Click on **Create Operation Schema** and enter **Open.**

The operation includes *Account* and changes the state and declares inputs - answer **YES** to these prompts from ZAPPA.

After answering yes to inputs a multiple selection menu appears listing the inputs you have created. Clicking on one turns the highlight on and off. Select both **new?** and **odl?** by highlighting them and then click on **OK.**

Answer **NO** to outputs.

se
presented with an operation schema template.

Preconditions are entered where the **NULL** is. Postconditions are entered after the curly bracket next to **Then.**

Position the cursor and erase the **NULL** as before. Open brackets and enter a ZAPPA translation of the first precondition:

$$new? \notin \text{dom}\ balance$$

You will have to refer to your function listing from now on.

After entering the first predicate type **And** and **return,** enter the second precondition and close brackets. Press the down arrow on the keyboard and the cursor should be positioned just after the curly bracket next to **Then.** Press **return** and **Tab** (the **Tab** key) and enter the first postconditon, which in Z is

$$balance' \quad = \quad balance \cup \{new? \mapsto 0\}$$

Translated this becomes

```
zequate (balance,
        zunion (balance,
                zmake_set1(
                zmake_map (new?, zeroZ))));
```

Note that

- after-state variables are not primed in ZAPPA - the first argument to **zequate** is **balance** not **balance**'.
- The **1** in **zmake_set1** indicates **one** element (the single maplet in this case).
- **zmake_map** forms a single maplet (in this case **new?** mapping to **zeroZ**)
- postconditions must end with a semi-colon.

Enter the first postcondition, translate the second and enter it also, noting that you have to use **zneg** (<u>neg</u> for negate) to negate <u>odl?</u>.

Rather than use the *Success* schema and *REPORT* free type for messages ZAPPA includes **zmessage** in templates allowing the user to enter appropriate operation and error messages between the quotes in

> **zmessage(" ");**

Position the cursor after the first quote mark and enter **Account Opened**. Do not press **Return**.

Once you are happy with your schema click on the small grey square in the top left hand corner of the Function Editor. Click on **Close** and then **Yes** to save the schema.

There are two error schemas associated with *Open*; *AlreadyExists* and *NegOdLimit*.

Taking *AlreadyExists* first, in the ANIMATOR click on **Create Error Schema** and enter **AlreadyExists.** *AlreadyExits* does include *Account* but does not change the state.

Enter the Function Editor, as before, to edit the error schema template.

AlreadyExists has one precondition

$$new? \in \mathrm{dom}\ balance$$

Erase **NULL** as before and enter the ZAPPA translation for the precondition.

AlreadyExists has no postconditions apart from an error message. Using the mouse or arrow keys position the cursor between the quotes in

> **zmessage (" ");**

and enter **account already exists** or some other appropriate message.

Save the schema as before.

Create *NegOdLimit* in the same way, noting that there are two preconditions to translate this time.

If you have not done so recently, check on the progress of the animation with OVERVIEW.

(c) Creating *ROpen*

We will now use schema calculus to define *ROpen* in the form

$$ROpen = Open \lor AlreadyExists \lor NegOdLimit$$

(remember, we are not using the success schema approach).

Click on **Create Robust Op Schema** and enter **ROpen**.

Enter the Editor to edit the robust operation template for *ROpen*.

Erase the **NULL** and open brackets.

In ZAPPA the simple schema calculus for defining robust operation schemas takes the form

```
If      (OpSchema()        Or
         ErrSchema1()      Or
         ErrSchema2()      Or
              ⋮
         ErrSchema n() )

Then TRUE
```

...

Noting the empty brackets after each schema name enter the appropriate schema calculus for **ROpen** and save the robust operation created.

Go to othe ANIMATION and note that a button for the robust operation has appeared.

6 Opening Accounts

Now let us open some accounts.

Because we have used a parachuted basic type, *ACC_No*, we must now decided what to use for account numbers. The simplest approach (and , I think, best approach from the point of view of animation) is to use straightforward numberd symbols, in this case **ac1**, **ac2**, **ac3** etc.

Click on *ROpen* and enter **ac1** as the new account number and **100** as the overdraft limit. Be patient, robust operations can be slow!

If all is well you should get the successful operation message **Account Opened**. Click on **OK**.

Now open accounts **ac2**, **ac3**, **ac4** and **ac5** with overdraft limits of *200*, *300*, *400* and *500* respectively.

Use **show var** to check that the accounts have been opened properly.

Now test for robustness:

Try opening an account that already exists. Try opening an account entering a negative overdraft limit.

[NOTE: If syntax errors are reported use **KTOOLS Function Edit SchemaName OK** to enter the function Editor to correct the mistake.]

Use **DICheck** to ensure that no data invariants have been violated.

7 Creating RDeposit

Create the input variables *acc?* and *amount?*.

Create the operation schema *Deposit*.
[NOTE: *balance acc?* translates as balance (acc?)]

Create the error schemas *NoSuchAccount* and *NotPosAmnt*.

Create the robust operation schema *RDeposit*.

Deposit some monies into accounts.

Test for robustness.

8 Creating *Withdraw* and *Close*

In ZAPPA it is possible to use an operation schema without having first crreated error schemas.

This is achieved by creating a *RobustOpSchema* using only the operation schema in the schema calculus.

Create	*RWithdraw*	=	*Withdraw* and
	RClose	=	*Close*.

[NOTE: Use the domain subtract version of *Close*.]

Withdraw monies and close accounts.

Once you have finished click on **QUIT** (do not save changes!).

ZAPPA Function Listing (Mathematical Toolkit) for Banking Spec Tutorial

ZAPPA	Z
zdom(function)	dom function
zdom_sub(set, function)	set function
zelement?(element,set)	element ∈ set
zequal?(item1,item2)	item1 = item2
zequate(item1,item2)	item1' = item2
zfor_all_01 (dummy,set,value or function1 or dummy,math_rel*,value or function2 or dummy)	∀ dummy : set • value or function1(dummy) or dummy math_rel, value or function2(dummy) or dummy
zg?(value1,value2)	value1 > value2
zge?(value1,value2)	value1 ≥ value2
zl?(value1,value2)	value1 < value2
zle?(value1,value2)	value1 ≤ value2
zmake_empty(set)	set' = ∅
zmake_map(item1,item2)	item1 ↦ item2
zmake_set1(element)	{element}
zminus(value1,value2)	value1 - value2
zneg(value)	- value
zno_change(state_var)	state_var' = state_var
znot_element?(element,set)	element ∉ set
zoverride(function1,function2)	function1 ⊕ function2
zplus(value1,value2)	value1 + value2
zunion(set1,set2)	set1 ∪ set2

*math_rel can be =, != (≠), >,<,> = (≥), or < = (≤)

```
    /************************************
     **** FUNCTION: xpara_type
     ************************************/
MakeFunction( xpara_type, [],
  {
  ResetValue( Global:Type );
  PostInputForm( "Enter name of type : ", Global:Type, name );
  If ( Instance?( Global:Type ) Or Class?( Global:Type ) )
    Then PostMessage( "This name has already been used." )
    Else If Member?( Global:TypesGiven, Global:Type )
         Then PostMessage( Global:Type # " is a given type." )
         Else {
             MakeInstance( Global:Type, ParaTypes );
             AppendToList( Global:TypesPara, Global:Type );
             PostMessage( Global:Type # " has been parachuted into the spec." );
             };
  } );

    /************************************
     **** FUNCTION: xdel_type
     ************************************/
MakeFunction( xdel_type, [],
  {
  If ( LengthList( Global:TypesPara ) == 0 And LengthList( Global:TypesFree )
       == 0 And LengthList( Global:TypesSchema )
       == 0 )
    Then PostMessage( "There are no user defined types." )
    Else {
       Global:Type = PostMenu( "Select type to delete", Global:TypesPara,
                  Global:TypesFree, Global:TypesSchema,
                  "* CANCEL *" );
       If Not( Global:Type #= "* CANCEL *" )
         Then If Member?( Global:TypesUsed, Global:Type )
              Then PostMessage( Global:Type # " has been used in the specification." )
              Else {
                  If Member?( Global:TypesPara, Global:Type )
                    Then {
                        DeleteInstance( Global:Type );
                        RemoveFromList( Global:TypesPara,
                           Global:Type );
                        PostMessage( Global:Type # " has been deleted." );
                        };
                  };
       };
  } );

    /************************************
     **** FUNCTION: xcreate_state
     ************************************/
MakeFunction( xcreate_state, [],
  {
  If Null?( Global:State )
    Then {
       PostInputForm( "Enter name of state schema : ", Global:State,
          name );
       If ( Class?( Global:State ) Or Instance?( Global:State ) )
         Then {
            PostMessage( " The name " # Global:State # " has already been used." );
            ResetValue( Global:State );
            }
        Else {
            MakeClass( Global:State, StateSchema );
            PostMessage( Global:State # " state schema box has been created." );
            };
        }
    Else PostMessage( "You already have a state schema." );
  } );

     /**************************************
      **** FUNCTION: xdel_state
      **************************************/
MakeFunction( xdel_state, [],
   {
   If Null?( Global:State )
    Then PostMessage( "There is no state schema" )
    Else If ( LengthList( Global:StateVars ) > 0 )
         Then PostMessage( "CANNOT DELETE STATE - There is a state variable" )
         Else If ( Not( Null?( SelectList( Global:Ops, op,
                        op:IncState ) ) )
                 Or Not( Null?( SelectList( Global:Errs,
                           err, err:IncState ) ) )
                 Or Not( Null?( Global:Init ) ) )
              Then PostMessage( "CANNOT DELETE STATE -
           There is a schema which includes the state" )
              Else {
                 Global:YesNo = PostMenu( "Delete " # Global:State
                               # " ?", YES,
                          NO );
                 If ( Global:YesNo #= YES )
                   Then {
                      PostMessage( Global:State # " deleted" );
                      DeleteClass( Global:State );
                      ResetValue( Global:State );
                      };
                 };
  } );

     /**************************************
      **** FUNCTION: xcreate_op
      **************************************/
 MakeFunction( xcreate_op, [],
   {
   xcreate_schema( O );
   } );

     /**************************************
      **** FUNCTION: xdel_op
      **************************************/
 MakeFunction( xdel_op, [],
   {
   If ( LengthList( Global:Ops ) == 0 )
     Then PostMessage( "There are no operation schemas" )
     Else {
        Global:Varname = PostMenu( "Select operation schema to delete",
                    Global:Ops, "* CANCEL *" );
      If Not( Global:Varname #= "* CANCEL *" )
        Then {
            Global:YesNo = PostMenu( "Delete " # Global:Varname
                                    # " ?", YES, NO );
            If ( Global:YesNo #= YES )
              Then {
                  PostMessage( Global:Varname # " deleted" );
                  DeleteClass( Global:Varname );
                  RemoveFromList( Global:Ops, Global:Varname );
                  DeleteFunction( Global:Varname );
                  };
            };
      };
  } );

    /************************************
     **** FUNCTION: xcreate_err
     ************************************/
MakeFunction( xcreate_err, [],
  {
  xcreate_schema( E );
  } );

    /************************************
     **** FUNCTION: xdel_err
     ************************************/
MakeFunction( xdel_err, [],
  {
  If ( LengthList( Global:Errs ) == 0 )
    Then PostMessage( "There are no error schemas" )
    Else {
        Global:Varname = PostMenu( "Select error schema to delete",
                        Global:Errs, "* CANCEL *" );
        If Not( Global:Varname #= "* CANCEL *" )
          Then {
              Global:YesNo = PostMenu( "Delete " # Global:Varname
                                      # " ?", YES, NO );
              If ( Global:YesNo #= YES )
                Then {
                    PostMessage( Global:Varname # " deleted" );
                    DeleteClass( Global:Varname );
                    RemoveFromList( Global:Errs, Global:Varname );
                    DeleteFunction( Global:Varname );
                    };
              };
        };
  } );

    /************************************
     **** FUNCTION: xcreate_rop
     ************************************/
MakeFunction( xcreate_rop, [],
  {
  ResetValue( Global:Varname );
  PostInputForm( "Create Robust Op", Global:Varname, "Enter name of robust op schema" );
  If ( Class?( Global:Varname ) Or Instance?( Global:Varname ) )
    Then PostMessage( "ERROR: " # Global:Varname # " has already been used" )
    Else {
      SetValue( Global:ROpsNum, Global:ROpsNum + +1 );
      AppendToList( Global:ROps, Global:Varname );
      MakeClass( Global:Varname, ROpSchemas );
      Let [ROp GetNthElem( Global:ROpsButs, Global:ROpsNum )]
         {
         SetValue( ROp:Title, Global:Varname );
         SetValue( ROp:Action, Global:Varname );
         ResetImage( ROp );
         ShowImage( ROp );
         MakeFunction( Global:Varname, [ ],
            {
            If NULL
             Then TRUE
             Else zmessage( "schema is not robust" );
            } );
         PostMessage( "Select Function from KTOOLS window, select edit then double click on "
                  # Global:Varname );
         ShowWindow( KTOOLS );
         };
      };
   } );

    /************************************
     ****  FUNCTION: xdel_rop
     ************************************/
MakeFunction( xdel_rop, [],
   {
   If ( LengthList( Global:ROps ) == 0 )
    Then PostMessage( "There are no robust operation schemas" )
    Else {
       Global:Varname = PostMenu( "Select robust operation schema to delete",
                  Global:ROps, "* CANCEL *" );
       If Not( Global:Varname #= "* CANCEL *" )
         Then {
            Global:YesNo = PostMenu( "Delete " # Global:Varname
                          # " ?", YES, NO );
            If ( Global:YesNo #= YES )
             Then {
                DeleteClass( Global:Varname );
                If ( GetElemPos( Global:ROps, Global:Varname )
                     != Global:ROpsNum )
                  Then {
                     Let [from GetElemPos( Global:ROps,
                              Global:Varname )]
                        [to Global:ROpsNum]
                        {
                        For x From from To to
                            Do {
                           Let [a GetNthElem( Global:ROpsButs,
                                  x )]
                             [b GetNthElem( Global:ROpsButs,
                                  x + +1 )]
                             {
                             SetValue( a:Title,
                                b:Title );
                             SetValue( a:Action,
                                b:Action );
                             ResetImage( a );
                        ResetImage( b );
                        };
                    };
                };
            };
        HideImage( GetNthElem( Global:ROpsButs,
                Global:ROpsNum ) );
        SetValue( Global:ROpsNum,
           Global:ROpsNum - 1 );
        PostMessage( Global:Varname # " deleted" );
        RemoveFromList( Global:ROps, Global:Varname );
        DeleteFunction( Global:Varname );
        };
    };
  };
} );

  /************************************
   **** FUNCTION: xcreate_schema
   ************************************/
MakeFunction( xcreate_schema, [x],
  {
  ResetValue( Global:Varname );
  If ( x #= O )
   Then PostInputForm( "Create Op Schema", Global:Varname, "Enter op schema name" )
   Else PostInputForm( " Create Error Schema", Global:Varname,
        "Enter error schema name" );
  If ( Class?( Global:Varname ) Or Instance?( Global:Varname ) )
   Then PostMessage( Global:Varname # " has already been used" )
   Else {
     If ( x #= O )
      Then {
         MakeClass( Global:Varname, OpSchemas );
         AppendToList( Global:Ops, Global:Varname );
         }
      Else {
         MakeClass( Global:Varname, ErrSchemas );
         AppendToList( Global:Errs, Global:Varname );
         };
     Let [schema Global:Varname]
        {
        If ( PostMenu( schema # " includes " # Global:State
                 # " ?", YES, NO ) #= YES )
         Then SetValue( schema:IncState, TRUE )
         Else SetValue( schema:IncState, FALSE );
        If ( schema:IncState #= TRUE )
         Then {
            If ( PostMenu( schema # " changes state ?",
                  YES, NO ) #= YES )
             Then SetValue( schema:ChngState, TRUE )
             Else SetValue( schema:ChngState, FALSE );
            };
        If ( x #= O )
         Then {
            If ( PostMenu( "Any inputs to declare for "
                     # schema # " ?", YES, NO )
                  #= YES )
             Then PostMultipleSelection(
                "Select input/s for " # schema,
                schema:Inputs, Global:Inputs );
          If ( PostMenu( "Any ouputs to declare for "
                    # schema # " ?", YES, NO )
               #= YES )
           Then PostMultipleSelection(
                "Select ouput/s for " # schema,
                schema:Outputs, Global:Outputs );
          };
       If ( x #= E )
        Then xschema00( schema )
        Else {
          If ( LengthList( schema:Inputs )
               == 0 And LengthList( schema:Outputs )
               == 0 )
           Then xschema00( schema )
           Else If ( LengthList( schema:Inputs )
                   == 1 And LengthList( schema:Outputs )
                   == 0 )
               Then xschema10( schema )
               Else If ( LengthList( schema:Inputs )
                       == 2 And LengthList( schema:Outputs )
                       == 0 )
                   Then xschema20( schema )
                   Else If ( LengthList( schema:Inputs )
                           == 3 And LengthList( schema:Outputs )
                           == 0 )
                       Then xschema30( schema )
                       Else If ( LengthList( schema:Inputs )
                               ==
                               1 And
                               LengthList( schema:Outputs )
                               ==
                               1 )
                           Then xschema11( schema )
                           Else xschema00( schema );
          };
       PostMessage( "Now choose Function and Edit from
the Kappa Tools window and select "
                  # schema # " to enter
schema  predicate" );
       ShowWindow( KTOOLS );
       };
     };
  } );

    /************************************
     **** FUNCTION: xschema00
     ************************************/
MakeFunction( xschema00, [schema],
  {
  MakeFunction( schema, [ ],
    {
    If NULL
     Then {
       zmessage( " " );
       zend( );
       TRUE;
```

```
            }
        Else {
            zend( );
            FALSE;
            };
      } );
  } );

     /************************************
      **** FUNCTION: xschema10
      ************************************/
MakeFunction( xschema10, [schema],
   {
   MakeFunction( schema, [ ],
      {
      zinput( GetNthElem( schema:Inputs, 1 ) );
      If NULL
        Then {
            zmessage( " " );
            zend( );
            TRUE;
            }
        Else {
            zend( );
            FALSE;
            };
      } );
  } );

     /************************************
      **** FUNCTION: xschema20
      ************************************/
MakeFunction( xschema20, [schema],
   {
   MakeFunction( schema, [ ],
      {
      zinput( GetNthElem( schema:Inputs, 1 ) );
      zinput( GetNthElem( schema:Inputs, 2 ) );
      If NULL
        Then {
            zmessage( " " );
            zend( );
            TRUE;
            }
        Else {
            zend( );
            FALSE;
            };
      } );
  } );

     /************************************
      **** FUNCTION: xschema30
      ************************************/
MakeFunction( xschema30, [schema],
   {
   MakeFunction( schema, [ ],
      {
```

```
    zinput( GetNthElem( schema:Inputs, 1 ) );
    zinput( GetNthElem( schema:Inputs, 2 ) );
    zinput( GetNthElem( schema:Inputs, 3 ) );
    If NULL
     Then {
        zmessage( " " );
        zend( );
        TRUE;
        }
     Else {
        zend( );
        FALSE;
        };
    } );
  } );

    /************************************
    **** FUNCTION: xschema11
    ************************************/
MakeFunction( xschema11, [schema],
  {
  MakeFunction( schema, [ ],
    {
    zinput( GetNthElem( schema:Inputs, 1 ) );
    If NULL
     Then {
        zoutput( GetNthElem( schema:Outputs, 1 ) );
        zmessage( " " );
        zend( );
        TRUE;
        }
     Else {
        zend( );
        FALSE;
        };
    } );
  } );

    /************************************
    **** FUNCTION: xcreate_di
    ************************************/
MakeFunction( xcreate_di, [],
  {
  If Function?( DICheck )
   Then PostError( "ERROR  data invariant already created" )
   Else {
      MakeFunction( DICheck, [ ],
         {
         If NULL
          Then zmessage( "data invariants ok" )
          Else zmessage( "data invariant error" );
         zend( );
         } );
      PostMessage( "Click on Function in KTOOLS window, select edit and double click on
DICheck. Then enter state schema predicate" );
      ShowImage( Button7 );
      SetValue( Global:DI, TRUE );
      ShowWindow( KTOOLS );
```

```
        };
  } );

     /************************************
      **** FUNCTION: xdel_di
      ************************************/
MakeFunction( xdel_di, [],
  {
  If Not( Global:DI )
   Then PostMessage( "There is no data invariant" )
   Else {
       Global:YesNo = PostMenu( "Delete data invariant ?", YES,
                   NO );
       If ( Global:YesNo #= YES )
        Then {
            DeleteFunction( DICheck );
            HideImage( Button7 );
            SetValue( Global:DI, FALSE );
            PostMessage( "Data invariant deleted" );
            };
       };
  } );

     /************************************
      **** FUNCTION: xcreate_init
      ************************************/
MakeFunction( xcreate_init, [],
  {
  If Not( Null?( Global:Init ) )
   Then PostMessage( "An initial state already exists" )
   Else {
       PostInputForm( "Create Initial State", Global:Init, "Enter initial state schema name" );
       If ( Class?( Global:Init ) Or Instance?( Global:Init ) )
        Then {
            PostMessage( Global:Init # " has already been used" );
            ResetValue( Global:Init );
            }
        Else {
            MakeClass( Global:Init, InitSchema );
            SetValue( Button6:Title, Global:Init );
            SetValue( Button6:Action, Global:Init );
            ResetImage( Button6 );
            ShowImage( Button6 );
            MakeFunction( Global:Init, [ ],
               {
               zmessage( "state initialised" );
               zend( );
               } );
            PostMessage( "Select Function from KTOOLS window and then select "
                     # Global:Init # " to edit. Then enter init schema predicate" );
            ShowWindow( KTOOLS );
            };
       };
  } );

     /************************************
      **** FUNCTION: xdel_init
      ************************************/
```

```
MakeFunction( xdel_init, [],
   {
   If Null?( Global:Init )
    Then PostMessage( "There is no initial state" )
    Else {
        Global:YesNo = PostMenu( "Delete " # Global:Init # " ?",
                    YES, NO );
        If ( Global:YesNo #= YES )
         Then {
            DeleteClass( Global:Init );
            DeleteFunction( Global:Init );
            ResetValue( Global:Init );
            HideImage( Button6 );
            PostMessage( Global:Init # " deleted" );
            };
        };
   } );

     /************************************
     **** FUNCTION: xsave
     ************************************/
MakeFunction( xsave, [],
   {
   PostInputForm( "Save As", Global:FileName, "Enter File Name ( with no extension.)" );
   PostBusy( ON, "Saving Specification. Please Wait." );
   OpenWriteFile( Global:FileName # .KAL );
   If Not( Null?( Global:State ) )
    Then WriteClass( Global:State );
   WriteLine( "SetValue ( Global:State, ", Global:State, " );" );
   For temp From 1 To Global:ROpsNum
        Do {
       WriteLine( );
       WriteLine( "SetValue ( ", GetNthElem( Global:ROpsButs,
                          temp ), ":Title, ",
          GetValue( GetNthElem( Global:ROpsButs, temp ),
             Title ), " ) ;" );
       WriteLine( );
       WriteLine( "SetValue ( ", GetNthElem( Global:ROpsButs,
                          temp ), ":Action, ",
          GetValue( GetNthElem( Global:ROpsButs, temp ),
             Action ), " ) ;" );
       WriteLine( );
       WriteLine( "ResetImage ( ", GetNthElem( Global:ROpsButs,
                           temp ), " ) ;" );
       WriteLine( );
       WriteLine( "ShowImage ( ", GetNthElem( Global:ROpsButs,
                          temp ), " ) ;" );
       WriteLine( );
       };
   WriteLine( "SetValue ( Global:ROpsNum, ", Global:ROpsNum, " ) ;" );
   WriteLine( );
   xsave_schema( ROps );
   xsave_schema( Errs );
   xsave_schema( Ops );
   xsave_var( StateVars );
   xsave_var( Inputs );
   xsave_var( Outputs );
   xsave_var( Locals );
```

```
If Not( Null?( Global:Init ) )
  Then xsave_init(  );
If Global:DI
  Then xsave_di(  );
xsave_type( TypesPara );
xsave_type( TypesFree );
xsave_type( Type.Schema );
EnumList( Global:TypesUsed, temp, WriteLine( "AppendToList ( Global:TypesUsed, ",
                                temp, " );" ) )
```

```
    /************************************
     **** FUNCTION: xpara_type
     ************************************/
MakeFunction( xpara_type, [],
   {
   ResetValue( Global:Type );
   PostInputForm( "Enter name of type : ", Global:Type, name );
   If ( Instance?( Global:Type ) Or Class?( Global:Type ) )
     Then PostMessage( "This name has already been used." )
     Else If Member?( Global:TypesGiven, Global:Type )
          Then PostMessage( Global:Type # " is a given type." )
          Else {
              MakeInstance( Global:Type, ParaTypes );
              AppendToList( Global:TypesPara, Global:Type );
              PostMessage( Global:Type # " has been parachuted into the spec." );
              };
   } );

    /************************************
     **** FUNCTION: xdel_type
     ************************************/
MakeFunction( xdel_type, [],
   {
   If ( LengthList( Global:TypesPara ) == 0 And LengthList( Global:TypesFree )
        == 0 And LengthList( Global:TypesSchema )
        == 0 )
     Then PostMessage( "There are no user defined types." )
     Else {
        Global:Type = PostMenu( "Select type to delete", Global:TypesPara,
                     Global:TypesFree, Global:TypesSchema,
                     "* CANCEL *" );
        If Not( Global:Type #= "* CANCEL *" )
          Then If Member?( Global:TypesUsed, Global:Type )
               Then PostMessage( Global:Type # " has been used in the specification." )
               Else {
                   If Member?( Global:TypesPara, Global:Type )
                     Then {
                        DeleteInstance( Global:Type );
                        RemoveFromList( Global:TypesPara,
                           Global:Type );
                        PostMessage( Global:Type # " has been deleted." );
                        };
                   };
        };
   } );

    /************************************
     **** FUNCTION: xcreate_state
     ************************************/
MakeFunction( xcreate_state, [],
   {
   If Null?( Global:State )
     Then {
        PostInputForm( "Enter name of state schema : ", Global:State,
           name );
        If ( Class?( Global:State ) Or Instance?( Global:State ) )
          Then {
            PostMessage( " The name " # Global:State # " has already been used." );
            ResetValue( Global:State );
            }
        Else {
            MakeClass( Global:State, StateSchema );
            PostMessage( Global:State # " state schema box has been created." );
            };
        }
    Else PostMessage( "You already have a state schema." );
  } );

    /***********************************
     **** FUNCTION: xdel_state
     ***********************************/
MakeFunction( xdel_state, [],
  {
  If Null?( Global:State )
   Then PostMessage( "There is no state schema" )
   Else If ( LengthList( Global:StateVars ) > 0 )
        Then PostMessage( "CANNOT DELETE STATE - There is a state variable" )
        Else If ( Not( Null?( SelectList( Global:Ops, op,
                          op:IncState ) ) )
                Or Not( Null?( SelectList( Global:Errs,
                            err, err:IncState ) ) )
                Or Not( Null?( Global:Init ) ) )
            Then PostMessage( "CANNOT DELETE STATE -
          There is a schema which includes the state" )
            Else {
               Global:YesNo = PostMenu( "Delete " # Global:State
                                 # " ?", YES,
                             NO );
               If ( Global:YesNo #= YES )
                 Then {
                    PostMessage( Global:State # " deleted" );
                    DeleteClass( Global:State );
                    ResetValue( Global:State );
                    };
               };
  } );

    /***********************************
     **** FUNCTION: xcreate_op
     ***********************************/
MakeFunction( xcreate_op, [],
  {
  xcreate_schema( O );
  } );

    /***********************************
     **** FUNCTION: xdel_op
     ***********************************/
MakeFunction( xdel_op, [],
  {
  If ( LengthList( Global:Ops ) == 0 )
   Then PostMessage( "There are no operation schemas" )
   Else {
      Global:Varname = PostMenu( "Select operation schema to delete",
                  Global:Ops, "* CANCEL *" );
```

```
       If Not( Global:Varname #= "* CANCEL *" )
         Then {
            Global:YesNo = PostMenu( "Delete " # Global:Varname
                                # " ?", YES, NO );
            If ( Global:YesNo #= YES )
              Then {
                 PostMessage( Global:Varname # " deleted" );
                 DeleteClass( Global:Varname );
                 RemoveFromList( Global:Ops, Global:Varname );
                 DeleteFunction( Global:Varname );
                 };
            };
       };
   } );

    /***********************************
     **** FUNCTION: xcreate_err
     ***********************************/
MakeFunction( xcreate_err, [],
   {
   xcreate_schema( E );
   } );

    /***********************************
     **** FUNCTION: xdel_err
     ***********************************/
MakeFunction( xdel_err, [],
   {
   If ( LengthList( Global:Errs ) == 0 )
     Then PostMessage( "There are no error schemas" )
     Else {
        Global:Varname = PostMenu( "Select error schema to delete",
                    Global:Errs, "* CANCEL *" );
        If Not( Global:Varname #= "* CANCEL *" )
          Then {
             Global:YesNo = PostMenu( "Delete " # Global:Varname
                                 # " ?", YES, NO );
             If ( Global:YesNo #= YES )
               Then {
                  PostMessage( Global:Varname # " deleted" );
                  DeleteClass( Global:Varname );
                  RemoveFromList( Global:Errs, Global:Varname );
                  DeleteFunction( Global:Varname );
                  };
             };
        };
   } );

    /***********************************
     **** FUNCTION: xcreate_rop
     ***********************************/
MakeFunction( xcreate_rop, [],
   {
   ResetValue( Global:Varname );
   PostInputForm( "Create Robust Op", Global:Varname, "Enter name of robust op schema" );
   If ( Class?( Global:Varname ) Or Instance?( Global:Varname ) )
     Then PostMessage( "ERROR: " # Global:Varname # " has already been used" )
     Else {
      SetValue( Global:ROpsNum, Global:ROpsNum + +1 );
      AppendToList( Global:ROps, Global:Varname );
      MakeClass( Global:Varname, ROpSchemas );
      Let [ROp GetNthElem( Global:ROpsButs, Global:ROpsNum )]
         {
         SetValue( ROp:Title, Global:Varname );
         SetValue( ROp:Action, Global:Varname );
         ResetImage( ROp );
         ShowImage( ROp );
         MakeFunction( Global:Varname, [ ],
            {
            If NULL
             Then TRUE
             Else zmessage( "schema is not robust" );
            } );
         PostMessage( "Select Function from KTOOLS window, select edit then double click on "
                   # Global:Varname );
         ShowWindow( KTOOLS );
         };
      };
   } );

    /************************************
     **** FUNCTION: xdel_rop
     ************************************/
MakeFunction( xdel_rop, [],
   {
   If ( LengthList( Global:ROps ) == 0 )
    Then PostMessage( "There are no robust operation schemas" )
    Else {
       Global:Varname = PostMenu( "Select robust operation schema to delete",
                  Global:ROps, "* CANCEL *" );
       If Not( Global:Varname #= "* CANCEL *" )
         Then {
            Global:YesNo = PostMenu( "Delete " # Global:Varname
                           # " ?", YES, NO );
            If ( Global:YesNo #= YES )
             Then {
                DeleteClass( Global:Varname );
                If ( GetElemPos( Global:ROps, Global:Varname )
                      != Global:ROpsNum )
                  Then {
                     Let [from GetElemPos( Global:ROps,
                              Global:Varname )]
                        [to Global:ROpsNum]
                        {
                        For x From from To to
                            Do {
                           Let [a GetNthElem( Global:ROpsButs,
                                  x )]
                               [b GetNthElem( Global:ROpsButs,
                                  x + +1 )]
                               {
                               SetValue( a:Title,
                                  b:Title );
                               SetValue( a:Action,
                                  b:Action );
                               ResetImage( a );
                        ResetImage( b );
                        };
                      };
                    };
                  };
              HideImage( GetNthElem( Global:ROpsButs,
                      Global:ROpsNum ) );
              SetValue( Global:ROpsNum,
                 Global:ROpsNum - 1 );
              PostMessage( Global:Varname # " deleted" );
              RemoveFromList( Global:ROps, Global:Varname );
              DeleteFunction( Global:Varname );
              };
          };
      };
  } );

    /***********************************
     **** FUNCTION: xcreate_schema
     ***********************************/
MakeFunction( xcreate_schema, [x],
  {
  ResetValue( Global:Varname );
  If ( x #= O )
   Then PostInputForm( "Create Op Schema", Global:Varname, "Enter op schema name" )
   Else PostInputForm( " Create Error Schema", Global:Varname,
        "Enter error schema name" );
  If ( Class?( Global:Varname ) Or Instance?( Global:Varname ) )
   Then PostMessage( Global:Varname # " has already been used" )
   Else {
      If ( x #= O )
       Then {
          MakeClass( Global:Varname, OpSchemas );
          AppendToList( Global:Ops, Global:Varname );
          }
       Else {
          MakeClass( Global:Varname, ErrSchemas );
          AppendToList( Global:Errs, Global:Varname );
          };
      Let [schema Global:Varname]
        {
        If ( PostMenu( schema # " includes " # Global:State
                  # " ?", YES, NO ) #= YES )
         Then SetValue( schema:IncState, TRUE )
         Else SetValue( schema:IncState, FALSE );
        If ( schema:IncState #= TRUE )
         Then {
            If ( PostMenu( schema # " changes state ?",
                  YES, NO ) #= YES )
             Then SetValue( schema:ChngState, TRUE )
             Else SetValue( schema:ChngState, FALSE );
            };
        If ( x #= O )
         Then {
            If ( PostMenu( "Any inputs to declare for "
                      # schema # " ?", YES, NO )
                  #= YES )
             Then PostMultipleSelection(
                "Select input/s for " # schema,
                schema:Inputs, Global:Inputs );
           If ( PostMenu( "Any ouputs to declare for "
                    # schema # " ?", YES, NO )
                #= YES )
            Then PostMultipleSelection(
                "Select ouput/s for " # schema,
                schema:Outputs, Global:Outputs );
           };
        If ( x #= E )
         Then xschema00( schema )
         Else {
           If ( LengthList( schema:Inputs )
                == 0 And LengthList( schema:Outputs )
                == 0 )
            Then xschema00( schema )
            Else If ( LengthList( schema:Inputs )
                    == 1 And LengthList( schema:Outputs )
                    == 0 )
               Then xschema10( schema )
               Else If ( LengthList( schema:Inputs )
                       == 2 And LengthList( schema:Outputs )
                       == 0 )
                  Then xschema20( schema )
                  Else If ( LengthList( schema:Inputs )
                          == 3 And LengthList( schema:Outputs )
                          == 0 )
                     Then xschema30( schema )
                     Else If ( LengthList( schema:Inputs )
                             ==
                             1 And
                             LengthList( schema:Outputs )
                             ==
                             1 )
                        Then xschema11( schema )
                        Else xschema00( schema );
           };
        PostMessage( "Now choose Function and Edit from
the Kappa Tools window and select "
                # schema # " to enter
schema  predicate" );
        ShowWindow( KTOOLS );
        };
      };
  } );

   /************************************
    **** FUNCTION: xschema00
    *************************************/
MakeFunction( xschema00, [schema],
  {
  MakeFunction( schema, [ ],
    {
    If NULL
     Then {
       zmessage( " " );
       zend( );
       TRUE;
```

```
          }
       Else {
          zend( );
          FALSE;
          };
     } );
  } );

    /************************************
     **** FUNCTION: xschema10
     ************************************/
MakeFunction( xschema10, [schema],
  {
  MakeFunction( schema, [ ],
     {
     zinput( GetNthElem( schema:Inputs, 1 ) );
     If NULL
       Then {
          zmessage( " " );
          zend( );
          TRUE;
          }
       Else {
          zend( );
          FALSE;
          };
     } );
  } );

    /************************************
     **** FUNCTION: xschema20
     ************************************/
MakeFunction( xschema20, [schema],
  {
  MakeFunction( schema, [ ],
     {
     zinput( GetNthElem( schema:Inputs, 1 ) );
     zinput( GetNthElem( schema:Inputs, 2 ) );
     If NULL
       Then {
          zmessage( " " );
          zend( );
          TRUE;
          }
       Else {
          zend( );
          FALSE;
          };
     } );
  } );

    /************************************
     **** FUNCTION: xschema30
     ************************************/
MakeFunction( xschema30, [schema],
  {
  MakeFunction( schema, [ ],
     {
```

```
    zinput( GetNthElem( schema:Inputs, 1 ) );
    zinput( GetNthElem( schema:Inputs, 2 ) );
    zinput( GetNthElem( schema:Inputs, 3 ) );
    If NULL
     Then {
        zmessage( " " );
        zend( );
        TRUE;
        }
     Else {
        zend( );
        FALSE;
        };
    } );
  } );

    /*************************************
     **** FUNCTION: xschema11
     *************************************/
MakeFunction( xschema11, [schema],
  {
  MakeFunction( schema, [ ],
    {
    zinput( GetNthElem( schema:Inputs, 1 ) );
    If NULL
     Then {
        zoutput( GetNthElem( schema:Outputs, 1 ) );
        zmessage( " " );
        zend( );
        TRUE;
        }
     Else {
        zend( );
        FALSE;
        };
    } );
  } );

    /*************************************
     **** FUNCTION: xcreate_di
     *************************************/
MakeFunction( xcreate_di, [],
  {
  If Function?( DICheck )
   Then PostError( "ERROR  data invariant already created" )
   Else {
       MakeFunction( DICheck, [ ],
          {
          If NULL
           Then zmessage( "data invariants ok" )
           Else zmessage( "data invariant error" );
          zend( );
          } );
       PostMessage( "Click on Function in KTOOLS window, select edit and double click on
DICheck. Then enter state schema predicate" );
       ShowImage( Button7 );
       SetValue( Global:DI, TRUE );
       ShowWindow( KTOOLS );
```

```
        };
  } );

    /************************************
     **** FUNCTION: xdel_di
     ************************************/
MakeFunction( xdel_di, [],
  {
  If Not( Global:DI )
   Then PostMessage( "There is no data invariant" )
   Else {
      Global:YesNo = PostMenu( "Delete data invariant ?", YES,
                  NO );
      If ( Global:YesNo #= YES )
       Then {
          DeleteFunction( DICheck );
          HideImage( Button7 );
          SetValue( Global:DI, FALSE );
          PostMessage( "Data invariant deleted" );
          };
      };
  } );

    /************************************
     **** FUNCTION: xcreate_init
     ************************************/
MakeFunction( xcreate_init, [],
  {
  If Not( Null?( Global:Init ) )
   Then PostMessage( "An initial state already exists" )
   Else {
      PostInputForm( "Create Initial State", Global:Init, "Enter initial state schema name" );
      If ( Class?( Global:Init ) Or Instance?( Global:Init ) )
       Then {
          PostMessage( Global:Init # " has already been used" );
          ResetValue( Global:Init );
          }
       Else {
          MakeClass( Global:Init, InitSchema );
          SetValue( Button6:Title, Global:Init );
          SetValue( Button6:Action, Global:Init );
          ResetImage( Button6 );
          ShowImage( Button6 );
          MakeFunction( Global:Init, [ ],
             {
             zmessage( "state initialised" );
             zend( );
             } );
          PostMessage( "Select Function from KTOOLS window and then select "
                  # Global:Init # " to edit. Then enter init schema predicate" );
          ShowWindow( KTOOLS );
          };
      };
  } );

    /************************************
     **** FUNCTION: xdel_init
     ************************************/
```

```
MakeFunction( xdel_init, [],
   {
   If Null?( Global:Init )
     Then PostMessage( "There is no initial state" )
     Else {
         Global:YesNo = PostMenu( "Delete " # Global:Init # " ?",
                        YES, NO );
         If ( Global:YesNo #= YES )
           Then {
               DeleteClass( Global:Init );
               DeleteFunction( Global:Init );
               ResetValue( Global:Init );
               HideImage( Button6 );
               PostMessage( Global:Init # " deleted" );
               };
         };
   } );

     /************************************
      **** FUNCTION: xsave
      ************************************/
MakeFunction( xsave, [],
   {
   PostInputForm( "Save As", Global:FileName, "Enter File Name ( with no extension.)" );
   PostBusy( ON, "Saving Specification. Please Wait." );
   OpenWriteFile( Global:FileName # .KAL );
   If Not( Null?( Global:State ) )
     Then WriteClass( Global:State );
   WriteLine( "SetValue ( Global:State, ", Global:State, " );" );
   For temp From 1 To Global:ROpsNum
          Do {
         WriteLine( );
         WriteLine( "SetValue ( ", GetNthElem( Global:ROpsButs,
                             temp ), ":Title, ",
            GetValue( GetNthElem( Global:ROpsButs, temp ),
               Title ), " ) ;" );
         WriteLine( );
         WriteLine( "SetValue ( ", GetNthElem( Global:ROpsButs,
                             temp ), ":Action, ",
            GetValue( GetNthElem( Global:ROpsButs, temp ),
               Action ), " ) ;" );
         WriteLine( );
         WriteLine( "ResetImage ( ", GetNthElem( Global:ROpsButs,
                              temp ), " ) ;" );
         WriteLine( );
         WriteLine( "ShowImage ( ", GetNthElem( Global:ROpsButs,
                             temp ), " ) ;" );
         WriteLine( );
         };
    WriteLine( "SetValue ( Global:ROpsNum, ", Global:ROpsNum, " ) ;" );
    WriteLine( );
    xsave_schema( ROps );
    xsave_schema( Errs );
    xsave_schema( Ops );
    xsave_var( StateVars );
    xsave_var( Inputs );
    xsave_var( Outputs );
    xsave_var( Locals );
```

```
If Not( Null?( Global:Init ) )
  Then xsave_init( );
If Global:DI
  Then xsave_di( );
xsave_type( TypesPara );
xsave_type( TypesFree );
xsave_type( TypesSchema );
EnumList( Global:TypesUsed, temp, WriteLine( "AppendToList ( Global:TypesUsed, ",
                              temp, " );" ) )
```

```
    /************************************
     **** FUNCTION: xequate10
     ************************************/
MakeFunction( xequate10, [Tup1 Tup2],
   {
   If ( Tup2:sort != 10 )
     Then PostError( "zequate - Illegal argument" )
     Else If Not( Tup1:Atype #= Tup2:Atype Or
                 ( Member?( Global:Numerics, Tup1:Atype ) And
                  Member?( Global:Numerics, Tup2:Atype ) ) )
         Then PostError( "zequate - Type missmatch" )
         Else {
             If Global:CheckConstrs
                Then If xnumvioln?( Tup1, Tup2 )
                   Then PostError( FormatValue( "zequate - This assignment would cause a numeric
constraint violation on %s.", Tup1 ) );
             SetValue( Tup1:Aval, Tup2:Aval );
             };
   } );

    /************************************
     **** FUNCTION: xequate11
     ************************************/
MakeFunction( xequate11, [Tup1 Tup2],
   {
   If ( Tup2:sort != 11 )
     Then PostError( "zequate - Illegal Argument." )
     Else If Not( ( Tup1:Atype #= Tup2:Atype Or
                 ( Member?( Global:Numerics, Tup1:Atype ) And
                  Member?( Global:Numerics, Tup2:Atype ) ) ) And
                 ( Tup1:Btype #= Tup2:Btype Or
                 ( Member?( Global:Numerics, Tup1:Btype ) And
                  Member?( Global:Numerics, Tup2:Btype ) ) ) )
         Then PostError( "zequate - type missmatch" )
         Else {
             If Global:CheckConstrs
                Then If xnumvioln?( Tup1, Tup2 )
                   Then PostError( FormatValue( "zequate - This assignment would cause a numeric
constraint violation on %s.", Tup1 ) );
             SetValue( Tup1:Aval, Tup2:Aval );
             SetValue( Tup1:Bval, Tup2:Bval );
             };
   } );

    /************************************
     **** FUNCTION: xequate12
     ************************************/
MakeFunction( xequate12, [Tup1 Tup2],
   {
   If ( ( Tup1:sort == 12 And Tup2:sort != 12 ) Or
      ( Tup1:sort == 13 And Tup2:sort != 13 ) Or
      ( Tup1:sort == 14 And Tup2:sort != 14 ) )
     Then PostError( "zequate - Illegal argument." )
     Else If Not( ( Tup1:Atype #= Tup2:Atype Or
                 ( Member?( Global:Numerics, Tup1:Atype ) And
                  Member?( Global:Numerics, Tup2:Atype ) ) ) And
                 ( Tup1:Btype #= Tup2:Btype Or
              ( Member?( Global:Numerics, Tup1:Btype ) And
                Member?( Global:Numerics, Tup2:Btype ) ) ) And
              ( Tup1:Ctype #= Tup2:Ctype Or
              ( Member?( Global:Numerics, Tup1:Ctype ) And
                Member?( Global:Numerics, Tup2:Ctype ) ) ) )
        Then PostError( "zequate - type missmatch" )
        Else {
           If Global:CheckConstrs
             Then If xnumvioln?( Tup1, Tup2 )
               Then PostError( FormatValue( "zequate - This assignment would cause a numeric
constraint violation on %s.", Tup1 ) );
           SetValue( Tup1:Aval, Tup2:Aval );
           SetValue( Tup1:Bval, Tup2:Bval );
           SetValue( Tup1:Cval, Tup2:Cval );
           };
  } );

    /************************************
     **** FUNCTION: xequate21
     ************************************/
MakeFunction( xequate21, [Set1 Set2],
  {
  If ( Set2:sort != 21 )
    Then PostError( "zequate - Illegal argument." )
    Else If Not( Set1:Atype #= Set2:Atype Or
               ( Member?( Global:Numerics, Set1:Atype ) And
                 Member?( Global:Numerics, Set2:Atype ) ) )
        Then PostError( "zequate - type missmatch" )
        Else {
           If Global:CheckConstrs
             Then If xnumvioln?( Set1, Set2 )
               Then PostError( FormatValue( "zequate - This assignment would cause a numeric
constraint violation on %s.", Set1 ) );
           SetValue( Set1:Aelems, Set2:Aelems );
           };
  } );

    /************************************
     **** FUNCTION: xequate22
     ************************************/
MakeFunction( xequate22, [Set1 Set2],
  {
  If ( Set2:sort != 22 And Set2:sort != 41 And Set2:sort != 71 )
    Then PostError( "zequate - Illegal argument." )
    Else If Not( ( Set1:Atype #= Set2:Atype Or
               ( Member?( Global:Numerics, Set1:Atype ) And
                 Member?( Global:Numerics, Set2:Atype ) ) ) And
               ( Set1:Btype #= Set2:Btype Or
               ( Member?( Global:Numerics, Set1:Btype ) And
                 Member?( Global:Numerics, Set2:Btype ) ) ) )
        Then PostError( "zequate - type missmatch" )
        Else {
           If Global:CheckConstrs
             Then {
               If xconstrvioln?( Set1:sort, Set2 )
                 Then PostError( "zequate - This assignment would cause a constraint violation on "
                             # Set1 # " of sort " # Set1:sort # . );
               If xnumvioln?( Set1, Set2 )
```

```
                Then PostError( FormatValue( "zequate - This assignment would cause a numeric
constraint violation on %s.", Set1 ) );
                };
            SetValue( Set1:Aelems, Set2:Aelems );
            SetValue( Set1:Belems, Set2:Belems );
            };
  } );

    /************************************
     **** FUNCTION: xequate23
     ************************************/
MakeFunction( xequate23, [Set1 Set2],
  {
  If ( ( Set1:sort == 23 And Set2:sort != 23 ) Or
    ( ( Set1:sort == 24 Or Set1:sort == 72 ) And
     ( Set2:sort != 24 And Set2:sort != 42 And Set2:sort != 72 ) ) Or
    ( ( Set1:sort == 25 Or Set1:sort == 82 ) And
     ( Set2:sort != 25 And Set2:sort != 82 ) ) )
   Then PostError( "zequate - Illegal argument." )
   Else If Not( ( Set1:Atype #= Set2:Atype Or
               ( Member?( Global:Numerics, Set1:Atype ) And
                Member?( Global:Numerics, Set2:Atype ) ) ) And
               ( Set1:Btype #= Set2:Btype Or
               ( Member?( Global:Numerics, Set1:Btype ) And
                Member?( Global:Numerics, Set2:Btype ) ) ) And
               ( Set1:Ctype #= Set2:Ctype Or
               ( Member?( Global:Numerics, Set1:Ctype ) And
                Member?( Global:Numerics, Set2:Ctype ) ) ) )
       Then PostError( "zequate - type missmatch" )
       Else {
           If Global:CheckConstrs
             Then {
                If xconstrvioln?( Set1:sort, Set2 )
                  Then PostError( "zequate - This assignment would cause a constraint violation on "
                                # Set1 # " of sort " # Set1:sort # . );
                If xnumvioln?( Set1, Set2 )
                  Then PostError( FormatValue( "zequate - This assignment would cause a numeric
constraint violation on %s.", Set1 ) );
                };
            SetValue( Set1:Aelems, Set2:Aelems );
            SetValue( Set1:Belems, Set2:Belems );
            SetValue( Set1:Celems, Set2:Celems );
            };
  } );

    /************************************
     **** FUNCTION: xequate41
     ************************************/
MakeFunction( xequate41, [Set1 Set2],
  {
  If ( ( Set1:sort == 41 ) And
     ( Set2:sort != 22 And Set2:sort != 41 And Set2:sort != 71 ) )
   Then PostError( "zequate - Illegal argument." )
   Else If Not( ( Set1:Atype #= Set2:Atype Or
               ( Member?( Global:Numerics, Set1:Atype ) And
                Member?( Global:Numerics, Set2:Atype ) ) ) And
               ( Set1:Btype #= Set2:Btype Or
               ( Member?( Global:Numerics, Set1:Btype ) And
               Member?( Global:Numerics, Set2:Btype ) ) ) )
         Then PostError( "zequate - type missmatch" )
         Else {
            If Global:CheckConstrs
              Then {
                 If xconstrvioln?( Set1:sort, Set2 )
                   Then PostError( "zequate - This assignment would cause a constraint violation on "
                              # Set1 # " of sort " # Set1:sort # . );
                 If xnumvioln?( Set1, Set2 )
                   Then PostError( FormatValue( "zequate - This assignment would cause a numeric
constraint violation on %s.", Set1 ) );
                 };
            ResetValue( Set1:Aelems );
            ResetValue( Set1:Belems );
            For count From 1 To LengthList( Set2:Aelems )

                Do {
                AppendToList( Set1:Aelems, count );
                AppendToList( Set1:Belems,
                   GetNthElem( Set2:Belems,
                      GetElemPos( Set2:Aelems, count ) ) );
                };
            };
   } );

    /*************************************
    **** FUNCTION: xequate42
    *************************************/
MakeFunction( xequate42, [Set1 Set2],
   {
   If ( ( Set1:sort == 42 ) And
      ( Set2:sort != 24 And Set2:sort != 42 And Set2:sort != 72 ) )
    Then PostError( "zequate - Illegal argument." )
    Else If Not( ( Set1:Atype #= Set2:Atype Or
               ( Member?( Global:Numerics, Set1:Atype ) And
                Member?( Global:Numerics, Set2:Atype ) ) ) And
               ( Set1:Btype #= Set2:Btype Or
               ( Member?( Global:Numerics, Set1:Btype ) And
                Member?( Global:Numerics, Set2:Btype ) ) ) And
               ( Set1:Ctype #= Set2:Ctype Or
               ( Member?( Global:Numerics, Set1:Ctype ) And
                Member?( Global:Numerics, Set2:Ctype ) ) ) ) )
         Then PostError( "zequate - type missmatch" )
         Else {
            If Global:CheckConstrs
              Then {
                 If xconstrvioln?( Set1:sort, Set2 )
                   Then PostError( "zequate - This assignment would cause a constraint violation on "
                              # Set1 # " of sort " # Set1:sort # . );
                 If xnumvioln?( Set1, Set2 )
                   Then PostError( FormatValue( "zequate - This assignment would cause a numeric
constraint violation on %s.", Set1 ) );
                 };
            ResetValue( Set1:Aelems );
            ResetValue( Set1:Belems );
            ResetValue( Set1:Celems );
            For count From 1 To LengthList( Set2:Aelems )
```

```
                Do {
                AppendToList( Set1:Aelems, count );
                AppendToList( Set1:Belems,
                   GetNthElem( Set2:Belems,
                      GetElemPos( Set2:Aelems, count ) ) );
                AppendToList( Set1:Celems,
                   GetNthElem( Set2:Celems,
                      GetElemPos( Set2:Aelems, count ) ) );
                };
            };
   } );

    /************************************
     **** FUNCTION: xintersect21
     ************************************/
MakeFunction( xintersect21, [Set1 Set2],
   {
   If ( Set2:sort != 21 )
    Then PostError( " zintersect - Illegal argument." )
    Else If Not( Set1:Atype #= Set2:Atype Or
                ( Member?( Global:Numerics, Set1:Atype ) And
                  Member?( Global:Numerics, Set2:Atype ) ) )
        Then PostError( "zintersect - type missmatch" )
        Else {
            Global:TempNum += 1;
            Let [t Temp # Global:TempNum]
               {
               xmake_temp( 21, t, Set1 );
               EnumList( Set1:Aelems, x, If Member?( Set2:Aelems, x )
                              Then AppendToList( t:Aelems, x ) );
               t;
               };
            };
   } );

    /************************************
     **** FUNCTION: xintersect22
     ************************************/
MakeFunction( xintersect22, [Set1 Set2],
   {
   If ( Set2:sort != 22 And Set2:sort != 41 And Set2:sort !=
        71 )
    Then PostError( "zintersect - Illegal argument." )
    Else If Not( ( Set1:Atype #= Set2:Atype Or
                ( Member?( Global:Numerics, Set1:Atype ) And
                  Member?( Global:Numerics, Set2:Atype ) ) ) And
                ( Set1:Btype #= Set2:Btype Or
                ( Member?( Global:Numerics, Set1:Btype ) And
                  Member?( Global:Numerics, Set2:Btype ) ) ) )
        Then PostError( "zintersect - type missmatch" )
        Else {
            Global:TempNum += 1;
            Let [t1 Temp # Global:TempNum]
               {
               xmake_temp( 22, t1, Set1 );
               Global:TempNum += 1;
               Let [t2 Temp # Global:TempNum]
                  {
```

```
                xmake_temp( 11, t2, Set1 );
                For count From 1 To LengthList( Set2:Aelems )

                    Do {
                    SetValue( t2:Aval, GetNthElem( Set2:Aelems,
                                count ) );
                    SetValue( t2:Bval, GetNthElem( Set2:Belems,
                                count ) );
                    If zelement?( t2, Set1 )
                     Then {
                        AppendToList( t1:Aelems,
                           t2:Aval );
                        AppendToList( t1:Belems,
                           t2:Bval );
                        };
                    };
                };
            t1;
            };
          };
  } );

    /*************************************
     **** FUNCTION: xintersect23
     *************************************/
MakeFunction( xintersect23, [Set1 Set2],
   {'
   If ( ( Set1:sort == 23 And Set2:sort != 23 ) Or
      ( ( Set1:sort == 24 Or Set1:sort == 42 Or Set1:sort == 72 ) And
        ( Set2:sort != 24 And  Set2:sort != 42 And Set2:sort != 72 ) ) Or
      ( ( Set1:sort == 25 Or Set1:sort == 81 ) And
        ( Set2:sort != 25 And Set2:sort != 81 ) ) )
     Then PostError( "zintersect - Illegal argument." )
     Else If Not( ( Set1:Atype #= Set2:Atype Or
                ( Member?( Global:Numerics, Set1:Atype ) And
                  Member?( Global:Numerics, Set2:Atype ) ) ) And
                ( Set1:Btype #= Set2:Btype Or
                ( Member?( Global:Numerics, Set1:Btype ) And
                  Member?( Global:Numerics, Set2:Btype ) ) ) And
                ( Set1:Ctype #= Set2:Ctype Or
                ( Member?( Global:Numerics, Set1:Ctype ) And
                  Member?( Global:Numerics, Set2:Ctype ) ) ) ) )
        Then PostError( "zintersect - type missmatch" )
        Else {
           Global:TempNum += 1;
           Let [t1 Temp # Global:TempNum]
              {
              xmake_temp( 23, t1, Set1 );
              If ( Set1:sort == 24 Or Set1:sort == 42
                    Or Set1:sort == 72 )
                Then t1:sort = 24;
              If ( Set1:sort == 25 Or Set1:sort == 82 )
                Then t1:sort = 25;
              Global:TempNum += 1;
              Let [t2 Temp # Global:TempNum]
                 {
                 xmake_temp( 12, t2, Set1 );
                 If ( Set1:sort == 24 Or Set1:sort ==
                42 Or Set1:sort == 72 )
              Then t2:sort = 13;
            If ( Set1:sort == 25 Or Set1:sort ==
                81 )
              Then t2:sort = 14;
            For count From 1 To LengthList( Set2:Aelems )

                Do {
                SetValue( t2:Aval, GetNthElem( Set2:Aelems,
                            count ) );
                SetValue( t2:Bval, GetNthElem( Set2:Belems,
                            count ) );
                SetValue( t2:Cval, GetNthElem( Set2:Celems,
                            count ) );
                If zelement?( t2, Set1 )
                  Then {
                    AppendToList( t1:Aelems,
                      t2:Aval );
                    AppendToList( t1:Belems,
                      t2:Bval );
                    AppendToList( t1:Celems,
                      t2:Cval );
                    };
                };
            };
          t1;
          };
        };
  } );

    /************************************
    **** FUNCTION: xmake11
    ************************************/
MakeFunction( xmake11, [var1 var2],
  {
  Global:TempNum += 1;
  Let [temp Temp # Global:TempNum]
    {
    MakeClass( temp, Temp );
    MakeSlot( temp:sort );
    SetSlotOption( temp:sort, VALUE_TYPE, NUMBER );
    SetValue( temp:sort, 11 );
    MakeSlot( temp:Atype );
    MakeSlot( temp:Aval );
    If Member?( Global:Numerics, var1:Atype )
      Then {
        SetSlotOption( temp:Aval, VALUE_TYPE, NUMBER );
        SetValue( temp:Atype, Z );
        }
      Else SetValue( temp:Atype, var1:Atype );
    SetValue( temp:Aval, var1:Aval );
    MakeSlot( temp:Btype );
    MakeSlot( temp:Bval );
    If Member?( Global:Numerics, var2:Atype )
      Then {
        SetSlotOption( temp:Bval, VALUE_TYPE, NUMBER );
        SetValue( temp:Btype, Z );
        }
      Else SetValue( temp:Btype, var2:Atype );
    SetValue( temp:Bval, var2:Aval );
    temp;
    };
  } );

    /************************************
    **** FUNCTION: xmake12
    ************************************/
MakeFunction( xmake12, [var1 var2 var3],
  {
  Global:TempNum += 1;
  Let [temp Temp # Global:TempNum]
    {
    MakeClass( temp, Temp );
    MakeSlot( temp:sort );
    SetSlotOption( temp:sort, VALUE_TYPE, NUMBER );
    SetValue( temp:sort, 12 );
    MakeSlot( temp:Atype );
    MakeSlot( temp:Aval );
    If Member?( Global:Numerics, var1:Atype )
      Then {
        SetSlotOption( temp:Aval, VALUE_TYPE, NUMBER );
        SetValue( temp:Atype, Z );
        }
      Else SetValue( temp:Atype, var1:Atype );
    SetValue( temp:Aval, var1:Aval );
    MakeSlot( temp:Btype );
    MakeSlot( temp:Bval );
    If Member?( Global:Numerics, var1:Btype )
      Then {
        SetSlotOption( temp:Bval, VALUE_TYPE, NUMBER );
        SetValue( temp:Btype, Z );
        }
      Else SetValue( temp:Btype, var1:Btype );
    SetValue( temp:Bval, var2:Aval );
    MakeSlot( temp:Ctype );
    MakeSlot( temp:Cval );
    If Member?( Global:Numerics, var1:Ctype )
      Then {
        SetSlotOption( temp:Cval, VALUE_TYPE, NUMBER );
        SetValue( temp:Ctype, Z );
        }
      Else SetValue( temp:Ctype, var1:Ctype );
    SetValue( temp:Cval, var3:Aval );
    temp;
    };
  } );

    /************************************
    **** FUNCTION: xmake13
    ************************************/
MakeFunction( xmake13, [var1 var2],
  {
  Global:TempNum += 1;
  Let [temp Temp # Global:TempNum]
    {
    MakeClass( temp, Temp );
    MakeSlot( temp:sort );
    SetSlotOption( temp:sort, VALUE_TYPE, NUMBER );
    SetValue( temp:sort, 13 );
    MakeSlot( temp:Atype );
    MakeSlot( temp:Aval );
    If Member?( Global:Numerics, var1:Atype )
      Then {
           SetSlotOption( temp:Aval, VALUE_TYPE, NUMBER );
           SetValue( temp:Atype, Z );
           }
      Else SetValue( temp:Atype, var1:Atype );
    SetValue( temp:Aval, var1:Aval );
    MakeSlot( temp:Btype );
    MakeSlot( temp:Bval );
    If Member?( Global:Numerics, var2:Atype )
      Then {
           SetSlotOption( temp:Bval, VALUE_TYPE, NUMBER );
           SetValue( temp:Btype, Z );
           }
      Else SetValue( temp:Btype, var2:Atype );
    SetValue( temp:Bval, var2:Aval );
    MakeSlot( temp:Ctype );
    MakeSlot( temp:Cval );
    If Member?( Global:Numerics, var2:Btype )
      Then {
           SetSlotOption( temp:Cval, VALUE_TYPE, NUMBER );
           SetValue( temp:Ctype, Z );
           }
      Else SetValue( temp:Ctype, var2:Btype );
    SetValue( temp:Cval, var2:Bval );
    temp;
    };
  } );

    /************************************
     **** FUNCTION: xmake14
     ************************************/
MakeFunction( xmake14, [var1 var2],
  {
  Global:TempNum += 1;
  Let [temp Temp # Global:TempNum]
    {
    MakeClass( temp, Temp );
    MakeSlot( temp:sort );
    SetSlotOption( temp:sort, VALUE_TYPE, NUMBER );
    SetValue( temp:sort, 13 );
    MakeSlot( temp:Atype );
    MakeSlot( temp:Aval );
    If Member?( Global:Numerics, var1:Atype )
      Then {
           SetSlotOption( temp:Aval, VALUE_TYPE, NUMBER );
           SetValue( temp:Atype, Z );
           }
      Else SetValue( temp:Atype, var1:Atype );
    SetValue( temp:Aval, var1:Aval );
    MakeSlot( temp:Btype );
    MakeSlot( temp:Bval );
    If Member?( Global:Numerics, var1:Btype )
```

```
      Then {
          SetSlotOption( temp:Bval, VALUE_TYPE, NUMBER );
          SetValue( temp:Btype, Z );
          }
      Else SetValue( temp:Btype, var1:Btype );
    SetValue( temp:Bval, var1:Bval );
    MakeSlot( temp:Ctype );
    MakeSlot( temp:Cval );
    If Member?( Global:Numerics, var2:Atype )
      Then {
          SetSlotOption( temp:Cval, VALUE_TYPE, NUMBER );
          SetValue( temp:Ctype, Z );
          }
      Else SetValue( temp:Ctype, var2:Atype );
    SetValue( temp:Cval, var2:Aval );
    temp;
    };
  } );

    /*************************************
    **** FUNCTION: xmake211
    *************************************/
MakeFunction( xmake211, [elem],
  {
  Global:TempNum += 1;
  Let [temp Temp # Global:TempNum]
    {
    xmake_temp( 21, temp, elem );
    AppendToList( temp:Aelems, elem:Aval );
    temp;
    };
  } );

    /*************************************
    **** FUNCTION: xmake221
    *************************************/
MakeFunction( xmake221, [elem],
  {
  Global:TempNum += 1;
  Let [temp Temp # Global:TempNum]
    {
    xmake_temp( 22, temp, elem );
    AppendToList( temp:Aelems, elem:Aval );
    AppendToList( temp:Belems, elem:Bval );
    temp;
    };
  } );

    /*************************************
    **** FUNCTION: xmake231
    *************************************/
MakeFunction( xmake231, [elem],
  {
  Global:TempNum += 1;
  Let [temp Temp # Global:TempNum]
    {
    xmake_temp( 23, temp, elem );
    If ( elem:sort == 13 )
```

```
      Then SetValue( temp:sort, 24 )
      Else If ( elem:sort == 14 )
            Then SetValue( temp:sort, 25 );
    AppendToList( temp:Aelems, elem:Aval );
    AppendToList( temp:Belems, elem:Bval );
    AppendToList( temp:Celems, elem:Cval );
    temp;
    };
  } );

    /************************************
     **** FUNCTION: xsetcomp01a
     ************************************/
MakeFunction( xsetcomp01a, [dum set val1 rel val2],
  {
  If ( set:sort != 21 )
   Then PostError( "ERROR  zset_comp_01 : illegal argument, " # set );
  If Not( set:Atype #= dum:Atype Or
         ( Member?( Global:Numerics, set:Atype ) And
          Member?( Global:Numerics, dum:Atype ) ) )
    Then PostError( "ERROR  zset_comp_01 : type missmatch, dum/set" );
  If ( val1:sort == 71 )
   Then xsetcomp01a1( dum, set, val1, rel, val2 )
   Else PostError( "zset_comp_01 : NOT YET AVAILABLE" );
  } );

    /************************************
     **** FUNCTION: xsetcomp01a1
     ************************************/
MakeFunction( xsetcomp01a1, [dum set val1 rel val2],
  {
  If Not( Member?( Global:Numerics, val1:Btype ) )
   Then PostError( "ERROR  zset_comp_01 : " # val1 # " does not have numeric output" );
  If znot_subset?( set, zdom( val1 ) )
   Then PostError( "ERROR  zset_comp_01 : " # val1 # " not defined for all values of "
                # dum );
  If ( Not( dum #= val2 ) And val2:sort == 10 )
   Then xsetcomp01a1a( set, val1, rel, val2 )
   Else PostError( "zset_comp_01 : NOT YET AVAILABLE or illegal argument/s" );
  } );

    /************************************
     **** FUNCTION: xsetcomp01a1a
     ************************************/
MakeFunction( xsetcomp01a1a, [set val1 rel val2],
  {
  If Not( Member?( Global:Numerics, val2:Atype ) )
   Then PostError( "ERROR  zset_comp_01 : " # val2 # " is not numeric" );
  If Null?( val2:Aval )
   Then PostError( "ERROR  zset_comp_01 : " # val2 # " does not have a value" );
  If ( rel #= < )
   Then xsetcomp01a1a1( set, val1, val2 )
   Else PostError( "zset_comp_01 : NOT YET AVAILABLE" );
  } );

    /************************************
     **** FUNCTION: xsetcomp01a1a1
     ************************************/
```

```
MakeFunction( xsetcomp01a1a1, [set val1 val2],
  {
  Global:TempNum += 1;
  Let [t Temp # Global:TempNum]
    {
    MakeClass( t, Temp );
    MakeSlot( t:sort );
    SetSlotOption( t:sort, VALUE_TYPE, NUMBER );
    SetValue( t:sort, 21 );
    MakeSlot( t:Atype );
    SetValue( t:Atype, set:Atype );
    MakeSlot( t:Aelems );
    SetSlotOption( t:Aelems, MULTIPLE );
    If Member?( Global:Numerics, t:Atype )
     Then SetSlotOption( t:Aelem, VALUE_TYPE, NUMBER );
    EnumList( set:Aelems, x, If ( GetNthElem( val1:Belems,
                              GetElemPos( val1:Aelems,
                                x ) ) < val2:Aval )
                          Then AppendToList( t:Aelems,
                                x ) );
    t;
    };
  });

    /************************************
     **** FUNCTION: xsubtract21
     ************************************/
MakeFunction( xsubtract21, [set1 set2],
  {
  If ( set2:sort != 21 )
   Then PostError( "zsubtract - Illegal argument." )
   Else If Not( set1:Atype #= set2:Atype Or
              ( Member?( Global:Numerics, set1:Atype ) And
                Member?( Global:Numerics, set2:Atype ) ) )
       Then PostError( " zsubtract - Type missmatch." )
       Else {
          Global:TempNum += 1;
          Let [temp Temp # Global:TempNun]
            {
            xmake_temp( 21, temp, set1 );
            EnumList( set1:Aelems, x, If Not( Member?( set2:Aelems, x ) )
                                Then AppendToList( temp:Aelems, x ) );
            temp;
            };
          };
  });

    /************************************
     **** FUNCTION: xsubtract22
     ************************************/
MakeFunction( xsubtract22, [Set1 Set2],
  {
  If ( Set2:sort != 22 And Set2:sort != 41 And Set2:sort !=
       71 )
   Then PostError( "zsubtract - Illegal argument." )
   Else If Not( ( Set1:Atype #= Set2:Atype Or
              ( Member?( Global:Numerics, Set1:Atype ) And
                Member?( Global:Numerics, Set2:Atype ) ) ) And
                ( Set1:Btype #= Set2:Btype Or
                ( Member?( Global:Numerics, Set1:Btype ) And
                 Member?( Global:Numerics, Set2:Btype ) ) ) )
         Then PostError( "zsubtract - type missmatch" )
         Else {
             Global:TempNum += 1;
             Let [t1 Temp # Global:TempNum]
                {
                xmake_temp( 22, t1, Set1 );
                Global:TempNum += 1;
                Let [t2 Temp # Global:TempNum]
                   {
                   xmake_temp( 11, t2, Set1 );
                   For count From 1 To LengthList( Set1:Aelems )

                          Do {
                          SetValue( t2:Aval, GetNthElem( Set1:Aelems,
                                         count ) );
                          SetValue( t2:Bval, GetNthElem( Set1:Belems,
                                         count ) );
                          If Not( zelement?( t2, Set2 ) )
                           Then {
                               AppendToList( t1:Aelems,
                                  t2:Aval );
                               AppendToList( t1:Belems,
                                  t2:Bval );
                               };
                          };
                   };
                t1;
                };
             };
   } );

    /************************************
     **** FUNCTION: xsubtract23
     ************************************/
MakeFunction( xsubtract23, [Set1 Set2],
   {
   If ( ( Set1:sort == 23 And Set2:sort != 23 ) Or
      ( ( Set1:sort == 24 Or Set1:sort == 42 Or Set1:sort == 72 ) And
       ( Set2:sort != 24 And Set2:sort != 42 And Set2:sort != 72 ) ) Or
      ( ( Set1:sort == 25 Or Set1:sort == 81 ) And
       ( Set2:sort != 25 And Set2:sort != 81 ) ) )
     Then PostError( "zsubtract - Illegal argument." )
     Else If Not( ( Set1:Atype #= Set2:Atype Or
                ( Member?( Global:Numerics, Set1:Atype ) And
                 Member?( Global:Numerics, Set2:Atype ) ) ) And
                ( Set1:Btype #= Set2:Btype Or
                ( Member?( Global:Numerics, Set1:Btype ) And
                 Member?( Global:Numerics, Set2:Btype ) ) ) And
                ( Set1:Ctype #= Set2:Ctype Or
                ( Member?( Global:Numerics, Set1:Ctype ) And
                 Member?( Global:Numerics, Set2:Ctype ) ) ) )
         Then PostError( "zsubtract - type missmatch" )
         Else {
             Global:TempNum += 1;
             Let [t1 Temp # Global:TempNum]
```

```
            {
            xmake_temp( 23, t1, Set1 );
            If ( Set1:sort == 24 Or Set1:sort == 42
                  Or Set1:sort == 72 )
              Then t1:sort = 24;
            If ( Set1:sort == 25 Or Set1:sort == 82 )
              Then t1:sort = 25;
            Global:TempNum += 1;
            Let [t2 Temp # Global:TempNum]
               {
               xmake_temp( 12, t2, Set1 );
               If ( Set1:sort == 24 Or Set1:sort ==
                     42 Or Set1:sort == 72 )
                 Then t2:sort = 13;
               If ( Set1:sort == 25 Or Set1:sort ==
                     81 )
                 Then t2:sort = 14;
               For count From 1 To LengthList( Set1:Aelems )

                     Do {
                     SetValue( t2:Aval, GetNthElem( Set1:Aelems,
                                     count ) );
                     SetValue( t2:Bval, GetNthElem( Set1:Belems,
                                     count ) );
                     SetValue( t2:Cval, GetNthElem( Set1:Celems,
                                     count ) );
                     If Not( zelement?( t2, Set2 ) )
                       Then {
                          AppendToList( t1:Aelems,
                             t2:Aval );
                          AppendToList( t1:Belems,
                             t2:Bval );
                          AppendToList( t1:Celems,
                             t2:Cval );
                          };
                     };
               };
            t1;
            };
         };
   } );

    /*************************************
     **** FUNCTION: xunion21
     *************************************/
MakeFunction( xunion21, [Set1 Set2],
   {
   If ( Set2:sort != 21 )
    Then PostError( " zunion - Illegal argument." )
    Else If Not( Set1:Atype #= Set2:Atype Or
                ( Member?( Global:Numerics, Set1:Atype ) And
                  Member?( Global:Numerics, Set2:Atype ) ) )
        Then PostError( "zunion - type missmatch" )
        Else {
           Global:TempNum += 1;
           Let [t Temp # Global:TempNum]
              {
              xmake_temp( 21, t, Set1 );
            AppendToList( t:Aelems, Set1:Aelems );
            EnumList( Set2:Aelems, x, If Not( Member?( t:Aelems, x ) )
                          Then AppendToList( t:Aelems, x ) );
            t;
            };
          };
  } );

    /************************************
     **** FUNCTION: xunion22
     ************************************/
MakeFunction( xunion22, [Set1 Set2],
  {
  If ( Set2:sort != 22 And Set2:sort != 41 And Set2:sort !=
       71 )
   Then PostError( "zsubtract - Illegal argument." )
   Else If Not( ( Set1:Atype #= Set2:Atype Or
              ( Member?( Global:Numerics, Set1:Atype ) And
               Member?( Global:Numerics, Set2:Atype ) ) ) And
              ( Set1:Btype #= Set2:Btype Or
              ( Member?( Global:Numerics, Set1:Btype ) And
               Member?( Global:Numerics, Set2:Btype ) ) ) )
       Then PostError( "zsubtract - type missmatch" )
       Else {
          Global:TempNum += 1;
          Let [t1 Temp # Global:TempNum]
            {
            xmake_temp( 22, t1, Set1 );
            SetValue( t1:Aelems, Set1:Aelems );
            SetValue( t1:Belems, Set1:Belems );
            Global:TempNum += 1;
            Let [t2 Temp # Global:TempNum]
              {
              xmake_temp( 11, t2, Set1 );
              For count From 1 To LengthList( Set2:Aelems )

                 Do {
                 SetValue( t2:Aval, GetNthElem( Set2:Aelems,
                             count ) );
                 SetValue( t2:Bval, GetNthElem( Set2:Belems,
                             count ) );
                 If Not( zelement?( t2, t1 ) )
                  Then {
                     AppendToList( t1:Aelems,
                        t2:Aval );
                     AppendToList( t :Belems,
                        t2:Bval );
                     };
                 };
              };
            t1;
            };
          };
  } );

    /*************************************
     **** FUNCTION: xunion23
     *************************************/
```

```
MakeFunction( xunion23, [Set1 Set2],
  {
  If ( ( Set1:sort == 23 And Set2:sort != 23 ) Or
     ( ( Set1:sort == 24 Or Set1:sort == 42 Or Set1:sort == 72 ) And
      ( Set2:sort != 24 And Set2:sort != 42 And Set2:sort != 72 ) ) Or
     ( ( Set1:sort == 25 Or Set1:sort == 81 ) And
      ( Set2:sort != 25 And Set2:sort != 81 ) ) )
   Then PostError( "zunion - Illegal argument." )
   Else If Not( ( Set1:Atype #= Set2:Atype Or
               ( Member?( Global:Numerics, Set1:Atype ) And
                 Member?( Global:Numerics, Set2:Atype ) ) ) And
               ( Set1:Btype #= Set2:Btype Or
               ( Member?( Global:Numerics, Set1:Btype ) And
                 Member?( Global:Numerics, Set2:Btype ) ) ) And
               ( Set1:Ctype #= Set2:Ctype Or
               ( Member?( Global:Numerics, Set1:Ctype ) And
                 Member?( Global:Numerics, Set2:Ctype ) ) ) ) )
      Then PostError( "zunion - type missmatch" )
      Else {
         Global:TempNum += 1;
         Let [t1 Temp # Global:TempNum]
            {
            xmake_temp( 23, t1, Set1 );
            If ( Set1:sort == 24 Or Set1:sort == 42
                  Or Set1:sort == 72 )
             Then t1:sort = 24;
            If ( Set1:sort == 25 Or Set1:sort == 82 )
             Then t1:sort = 25;
            SetValue( t1:Aelems, Set1:Aelems );
            SetValue( t1:Belems, Set1:Belems );
            SetValue( t1:Celems, Set1:Celems );
            Global:TempNum += 1;
            Let [t2 Temp # Global:TempNum]
               {
               xmake_temp( 12, t2, Set1 );
               If ( Set1:sort == 24 Or Set1:sort ==
                     42 Or Set1:sort == 72 )
                Then t2:sort = 13;
               If ( Set1:sort == 25 Or Set1:sort ==
                     81 )
                Then t2:sort = 14;
               For count From 1 To LengthList( Set2:Aelems )

                    Do {
                    SetValue( t2:Aval, GetNthElem( Set2:Aelems,
                                     count ) );
                    SetValue( t2:Bval, GetNthElem( Set2:Belems,
                                     count ) );
                    SetValue( t2:Cval, GetNthElem( Set2:Celems,
                                     count ) );
                    If Not( zelement?( t2, t1 ) )
                     Then {
                        AppendToList( t1:Aelems,
                          t2:Aval );
                        AppendToList( t1:Belems,
                          t2:Bval );
                        AppendToList( t1:Celems,
                          t2:Cval );
                        };
                    };
                };
            t1;
            };
        };
  } );

    /************************************
     **** FUNCTION: zcard
     ************************************/
MakeFunction( zcard, [set],
  {
  If Member?( Global:VarTupCodes, set:sort )
   Then PostError( "zcard- Illegal argument." )
   Else LengthList( set:Aelems );
  } );

    /************************************
     **** FUNCTION: zequate
     ************************************/
MakeFunction( zequate, [Item1 Item2],
  {
  If Not( Class?( Item1 ) )
   Then PostError( "Error : " # Item1 # " not recognised" )
   Else If Not( Slot?( Item1, sort ) )
      Then PostError( "Error : illegal argument, " # Item1 );
  If Not( Class?( Item2 ) )
   Then PostError( "Error : " # Item2 # " not recognised" )
   Else If Not( Slot?( Item2, sort ) )
      Then PostError( "Error : illegal argument, " # Item2 );
  If ( Item1:sort == 10 )
   Then xequate10( Item1, Item2 )
   Else If ( Item1:sort == 11 )
      Then xequate11( Item1, Item2 )
      Else If ( Item1:sort == 12 Or Item1:sort == 13 Or
            Item1:sort == 14 )
         Then xequate12( Item1, Item2 )
         Else If ( Item1:sort == 21 )
            Then xequate21( Item1, Item2 )
            Else If ( Item1:sort == 22 Or Item1:sort
                  == 71 )
               Then xequate22( Item1, Item2 )
               Else If ( Item1:sort == 23 Or
                     Item1:sort ==
                     24 Or Item1:sort
                     == 25 Or Item1:sort
                     == 72 Or Item1:sort
                     == 81 )
                  Then xequate23( Item1,
                        Item2 )
                  Else If ( Item1:sort
                        == 41 )
                     Then xequate41( Item1,
                           Item2 )
                     Else If ( Item1:sort
                           ==
                           42 )
```

```
                                     Then xequate42( Item1,
                                            Item2 )
                                     Else PostError( "zequate - Not yet available." );
   } );

     /**************************************
      **** FUNCTION: zintersect
      **************************************/
MakeFunction( zintersect, [set1 set2],
   {
   xvalid_var( set1 );
   xvalid_var( set2 );
   If Member?( Global:VarTupCodes, set1:sort )
    Then PostError( "Error  zintersect : " # set1 # " is not a set" );
   If Member?( Global:VarTupCodes, set2:sort )
    Then PostError( "Error  zintersect : " # set2 # " is not a set" );
   If ( set1:sort == 21 )
    Then xintersect21( set1, set2 )
    Else If ( set1:sort == 22 Or set1:sort == 41 Or set1:sort
              == 71 )
        Then xintersect22( set1, set2 )
        Else If ( set1:sort == 23 Or set1:sort == 24 Or set1:sort
                  == 25 Or set1:sort == 42 Or set1:sort
                  == 72 Or set1:sort == 81 )
            Then xintersect23( set1, set2 )
            Else PostError( "zintersect - Not yet available." );
   } );

     /**************************************
      **** FUNCTION: zmake_empty
      **************************************/
 MakeFunction( zmake_empty, [Set],
   {
   If Member?( Global:VarTupCodes, Set:sort )
    Then PostError( "ERROR zmake_empty - illegal argument" )
    Else {
       If ( Set:sort == 21 )
        Then {
           ClearList( Set:Aelems );
           Set;
           }
        Else If ( Set:sort == 22 Or Set:sort == 41 Or Set:sort
                  == 71 )
            Then {
               ClearList( Set:Aelems );
               ClearList( Set:Belems );
               Set;
               }
            Else If ( Set:sort == 24 Or Set:sort == 25
                      Or Set:sort == 42 Or Set:sort
                      == 72 Or Set:sort == 81 )
                Then {
                   ClearList( Set:Aelems );
                   ClearList( Set:Belems );
                   ClearList( Set:Celems );
                   Set;
                   }
                Else PostError( "zmake_empty - Not yet available" );
         };
   });

    /************************************
      **** FUNCTION: zmake_map
      ************************************/
MakeFunction( zmake_map, [var1 var2],
   {
   xvalid_var( var1 );
   xvalid_var( var2 );
   If ( var1:sort == 10 And var2:sort == 10 )
    Then xmake11( var1, var2 )
    Else If ( var1:sort == 10 And var2:sort == 11 )
         Then xmake13( var1, var2 )
         Else If ( var1:sort == 11 And var2:sort == 10 )
              Then xmake14( var1, var2 )
              Else PostError( "zmake_map - Not yet available" );
   });

    /************************************
      **** FUNCTION: zmake_set1
      ************************************/
 MakeFunction( zmake_set1, [elem],
   {
   If ( elem:sort == 10 )
    Then xmake211( elem )
    Else If ( elem:sort == 11 )
        Then xmake221( elem )
        Else If ( elem:sort == 12 Or elem:sort == 13 Or elem:sort
                 == 14 )
            Then xmake231( elem )
            Else PostError( "zmake_set1 - Not yet available" );
   });

    /************************************
      **** FUNCTION: zmake_triple
      ************************************/
 MakeFunction( zmake_triple, [var1 var2 var3],
   {
   xvalid_var( var1 );
   xvalid_var( var2 );
   xvalid_var( var3 );
   If ( var1:sort == 10 And var2:sort == 10 And var3:sort ==
        10 )
    Then xmake12( var1, var2, var3 )
    Else PostError( "zmake_map - Not yet available" );
    });

    /************************************
      **** FUNCTION: zminus
      ************************************/
 MakeFunction( zminus, [num1 num2],
    {
    If ( num1:sort != 10 Or num2:sort != 10 )
    Then PostError( "ERROR zminus : bad argument" )
    Else If Not( Member?( Global:Numerics, num1:Atype ) And
               Member?( Global:Numerics, num2:Atype ) )
        Then PostError( "ERROR zminus : type missmatch" )
        Else If ( Not( Number?( num1:Aval ) ) Or Not( Number?( num2:Aval ) ) )
            Then PostError( "ERROR zminus : non numeric argument" )
            Else {
               Global:TempNum += 1;
               Let [temp Temp # Global:TempNum]
                   {
                   MakeClass( temp, Temp );
                   MakeSlot( temp:sort, 10 );
                   SetSlotOption( temp:sort, VALUE_TYPE, NUMBER );
                   MakeSlot( temp:Atype, Z );
                   MakeSlot( temp:Aval, num1:Aval -
                                   num2:Aval );
                   SetSlotOption( temp:Aval, VALUE_TYPE,
                       NUMBER );
                   temp;
                   };
               };
    } );

      /*************************************
       **** FUNCTION: zmod
       *************************************/
 MakeFunction( zmod, [num1 num2],
    {
    If ( num1:sort != 10 Or num2:sort != 10 )
      Then PostError( "ERROR zminus : bad argument" )
      Else If Not( Member?( Global:Numerics, num1:Atype ) And
               Member?( Global:Numerics, num2:Atype ) )
        Then PostError( "ERROR zminus : type missmatch" )
        Else If ( Not( Number?( num1:Aval ) ) Or Not( Number?( num2:Aval ) ) )
            Then PostError( "ERROR zminus : non numeric argument" )
            Else {
               Global:TempNum += 1;
               Let [temp Temp # Global:TempNum]
                   {
                   MakeClass( temp, Temp );
                   MakeSlot( temp:sort, 10 );
                   SetSlotOption( temp:sort, VALUE_TYPE, NUMBER );
                   MakeSlot( temp:Atype, Z );
                   MakeSlot( temp:Aval, num1:Aval -
                                   num2:Aval
                                   * Floor( num1:Aval
                                           /
                                           num2:Aval ) );
                   SetSlotOption( temp:Aval, VALUE_TYPE,
                       NUMBER );
                   temp;
                   };
               };
    } );

      /*************************************
       **** FUNCTION: zmult
       *************************************/
 MakeFunction( zmult, [num1 num2],
    {
    If ( num1:sort != 10 Or num2:sort != 10 )
      Then PostError( "ERROR zminus : bad argument" )
    Else If Not( Member?( Global:Numerics, num1:Atype ) And
               Member?( Global:Numerics, num2:Atype ) )
        Then PostError( "ERROR zminus : type missmatch" )
        Else If ( Not( Number?( num1:Aval ) ) Or Not( Number?( num2:Aval ) ) )
            Then PostError( "ERROR zminus : non numeric argument" )
            Else {
               Global:TempNum += 1;
               Let [temp Temp # Global:TempNum]
                 {
                 MakeClass( temp, Temp );
                 MakeSlot( temp:sort, 10 );
                 SetSlotOption( temp:sort, VALUE_TYPE, NUMBER );
                 MakeSlot( temp:Atype, Z );
                 MakeSlot( temp:Aval, num1:Aval *
                               num2:Aval );
                 SetSlotOption( temp:Aval, VALUE_TYPE,
                    NUMBER );
                 temp;
                 };
               };
    } );

    /************************************
      **** FUNCTION: zneg
      ************************************/
 MakeFunction( zneg, [var],
    {'
    If ( var:sort != 10 )
    Then PostError( "ERROR zneg : bad argument" )
    Else {
       If Not( Number?( var:Aval ) )
        Then PostError( "ERROR zneg : argument must be numeric" )
        Else {
           Global:TempNum += 1;
           Let [temp Temp # Global:TempNum]
              {
              MakeClass( temp, Temp );
              MakeSlot( temp:sort, 10 );
              SetSlotOption( temp:sort, VALUE_TYPE, NUMBER );
              MakeSlot( temp:Atype, Z );
              MakeSlot( temp:Aval, Negative( var:Aval ) );
              SetSlotOption( temp:Aval, VALUE_TYPE, NUMBER );
              temp;
              };
           };
       };
    } );

    /************************************
      **** FUNCTION: zno_change
      ************************************/
 MakeFunction( zno_change, [Var],
    {} );

    /************************************
      **** FUNCTION: zplus
      ************************************/
 MakeFunction( zplus, [num1 num2],
  {
  If ( num1:sort != 10 Or num2:sort != 10 )
    Then PostError( "ERROR zplus : bad argument" )
    Else If Not( Member?( Global:Numerics, num1:Atype ) And
                 Member?( Global:Numerics, num2:Atype ) )
        Then PostError( "ERROR zplus : type missmatch" )
        Else If ( Not( Number?( num1:Aval ) ) Or Not( Number?( num2:Aval ) ) )
            Then PostError( "ERROR zplus : non numeric argument" )
            Else {
                Global:TempNum += 1;
                Let [temp Temp # Global:TempNum]
                  {
                  MakeClass( temp, Temp );
                  MakeSlot( temp:sort, 10 );
                  SetSlotOption( temp:sort, VALUE_TYPE, NUMBER );
                  MakeSlot( temp:Atype, Z );
                  MakeSlot( temp:Aval, num1:Aval +
                                num2:Aval );
                  SetSlotOption( temp:Aval, VALUE_TYPE,
                    NUMBER );
                  temp;
                  };
                };
  });

    /*************************************
    **** FUNCTION: zset_comp_01
    *************************************/
MakeFunction( zset_comp_01, [dum set val1 rel val2],
  {
  If Not( Member?( Global:Locals, dum ) )
    Then PostError( "ERROR  zset_comp_01 : " # dum # " is not local\dummy var" );
  If ( Not( IsAKindOf?( set, Temp ) ) And Not( IsAKindOf?( set,
                                 Global:State ) )
       And Not( IsAKindOf?( set, Locals ) ) And Not( IsAKindOf?( set,
                                        Inputs ) ) )
    Then PostError( "ERROR  zset_comp_01 : illegal argument, set" );
  If Not( Member?( Global:MathRels, rel ) )
    Then PostError( "ERROR  zset_comp_01 : " # rel # " is not logical relation" );
  If ( Not( Member?( Global:Locals, val1 ) ) And Not( IsAKindOf?( val1,
                                       zeros ) )
       And Not( Member?( Global:StateVars, val1 ) )
       And Not( Member?( Global:Inputs, val1 ) ) )
    Then PostError( "ERROR  zset_comp_01 : illegal argument, val1" );
  If ( Not( Member?( Global:Locals, val2 ) ) And Not( IsAKindOf?( val2,
                                       zeros ) )
       And Not( Member?( Global:StateVars, val2 ) )
       And Not( Member?( Global:Inputs, val2 ) ) )
    Then PostError( "ERROR  zset_comp_01 : illegal argument, val2" );
  If ( dum:sort == 10 )
    Then xsetcomp01a( dum, set, val1, rel, val2 )
    Else PostError( "zset_comp_01 : NOT YET AVAILABE" );
  });

    /*************************************
    **** FUNCTION: zsubtract
    *************************************/
MakeFunction( zsubtract, [set1 set2],
  {
  xvalid_var( set1 );
  xvalid_var( set2 );
  If Member?( Global:VarTupCodes, set1:sort )
   Then PostError( "Error  zsubtract : " # set1 # " is not a set" );
  If Member?( Global:VarTupCodes, set2:sort )
   Then PostError( "Error  zsubtract : " # set2 # " is not a set" );
  If ( set1:sort == 21 )
   Then xsubtract21( set1, set2 )
   Else If ( set1:sort == 22 Or set1:sort == 41 Or set1:sort
           == 71 )
       Then xsubtract22( set1, set2 )
       Else If ( set1:sort == 23 Or set1:sort == 24 Or set1:sort
               == 25 Or set1:sort == 42 Or set1:sort
               == 72 Or set1:sort == 81 )
           Then xsubtract23( set1, set2 )
           Else PostError( "zsubtract - Not yet available." );
  });

   /*************************************
    **** FUNCTION: zunion
    *************************************/
MakeFunction( zunion, [set1 set2],
  {
  xvalid_var( set1 );
  xvalid_var( set2 );
  If Member?( Global:VarTupCodes, set1:sort )
   Then PostError( "Error  zunion : " # set1 # " is not a set" );
  If Member?( Global:VarTupCodes, set2:sort )
   Then PostError( "Error  zunion : " # set2 # " is not a set" );
  If ( set1:sort == 21 )
   Then xunion21( set1, set2 )
   Else If ( set1:sort == 22 Or set1:sort == 41 Or set1:sort
           == 71 )
       Then xunion22( set1, set2 )
       Else If ( set1:sort == 23 Or set1:sort == 24 Or set1:sort
               == 25 Or set1:sort == 42 Or set1:sort
               == 72 Or set1:sort == 81 )
           Then xunion23( set1, set2 )
           Else PostError( "zunion - Not yet available." );
  });
```

```
    /************************************
     **** FUNCTION: xconcat41
     ************************************/
MakeFunction( xconcat41, [Seq1 Seq2],
{
If ( Seq1:sort != 41 And xconstrvioln?( 41, Seq1 ) )
  Then PostError( "zconcat - " # Seq1 # " is not a sequence." )
  Else If Not ( Seq2:sort == 41
              Or ( Member?( Global:Numerics, Seq2:Atype )
                  And Min( Seq2:Aelems ) >= 1
                  And ( Seq2:sort == 22 Or Seq2:sort == 71 ) ) )
        Then PostError( "zconcat - Illegal argument." )
        Else If ( Seq2:sort != 41 And xconstrvioln?( 41, Seq2 ) )
              Then PostError( "zconcat - " # Seq2 # " is not a sequence." )
              Else If Not ( ( Seq1:Btype #= Seq2:Btype ) Or
                          ( Member?( Global:Numerics, Seq1:Btype ) And
                           Member?( Global:Numerics, Seq2:Btype ) ) )
                    Then PostError( "zconcat - Type missmatch." )
                    Else {
                      Global:TempNum += 1;
                      Let [ temp Temp # Global:TempNum ]
                        {
                        xmake_temp( 22, temp, Seq1 );
                        SetValue( temp:sort, 41 );
                        For count From 1 To LengthList( Seq1:Aelems ) Do {
                          Let [ pos GetElemPos( Seq1:Aelems, count ) ]
                            {
                            AppendToList( temp:Aelems, count );
                            AppendToList( temp:Belems, GetNthElem( Seq1:Belems, pos ));
                            };
                          };
                        Let [ start LengthList( Seq1:Aelems ) ]
                            {
                            For count From 1 To LengthList( Seq2:Aelems ) Do {
                              Let [ pos GetElemPos( Seq2:Aelems, count ) ]
                                {
                                AppendToList( temp:Aelems, count + start );
                                AppendToList( temp:Belems, GetNthElem( Seq2:Belems,
pos ));
                                };
                              };
                            };
                        temp;
                        };
                      };
});

     /************************************
      **** FUNCTION: xcreate_func
      ************************************/
MakeFunction( xcreate_func, [varname x],
  {
  Global:Varstruct = PostMenu( "Select variable structure", Global:VarFns,
              "* CANCEL *" );
  If Not( Global:Varstruct #= "* CANCEL *" )
   Then {
     If ( Global:Varstruct #= "A -|-> B" )
```

```
        Then xcreate71( varname, x )
        Else If ( Global:Varstruct #= "A --|--> BxC" )
            Then xcreate72( varname, x )
            Else If ( Global:Varstruct #= "AxB --|--> C" )
                Then xcreate81( varname, x )
                Else {
                    PostMessage( "NOT YET IMPLEMENTED" );
                    xremove_var( varname, x );
                    };
        }
    Else {
        xremove_var( varname, x );
        PostMessage( varname # " deleted." );
        };
  } );

    /*************************************
     **** FUNCTION: xcreate_set
     *************************************/
MakeFunction( xcreate_set, [varname x],
  {
  Global:Varstruct = PostMenu( "Select variable structure", Global:VarSets,
               "* CANCEL *" );
  If Not( Global:Varstruct #= "* CANCEL *" )
    Then {
        If ( Global:Varstruct #= FA )
         Then xcreate21( varname, x )
         Else If ( Global:Varstruct #= "F ( AxB )" )
             Then xcreate22( varname, x )
             Else If ( Global:Varstruct #= "F ( AxBxC )" )
                 Then xcreate23( varname, x )
                 Else If ( Global:Varstruct #= "F ( Ax( BxC ))" )
                     Then xcreate24( varname, x )
                     Else If ( Global:Varstruct #=
                                 "F (( AxB )xC )" )
                         Then xcreate25( varname,
                              x )
                         Else {
                            xremove_var( varname,
                              x );
                            PostMessage( varname
                                      #
                                      " deleted." );
                            };
        }
    Else {
        xremove_var( varname, x );
        PostMessage( varname # " deleted." );
        };
  } );

    /*************************************
     **** FUNCTION: xcreate_var
     *************************************/
MakeFunction( xcreate_var, [x],
  {
  If ( x #= S And Null?( Global:State ) )
    Then PostMessage( "You have not created a state schema box." )
  Else {
     ResetValue( Global:Varname );
     PostInputForm( "Enter variable name :", Global:Varname,
        Name );
     If ( Class?( Global:Varname ) Or Instance?( Global:Varname ) )
      Then PostMessage( Global:Varname # " has already been used." )
      Else {
         Global:Sort = PostMenu( "What sort of variable ?",
                      Tuple, Set, Bag, Sequence,
                      Function, "* CANCEL *" );
         If Not( Global:Sort #= "* CANCEL *" )
          Then {
             If ( x #= L )
              Then {
                 MakeClass( Global:Varname, Locals );
                 AppendToList( Global:Locals, Global:Varname );
                 };
             If ( x #= S )
              Then {
                 MakeClass( Global:Varname, Global:State );
                 AppendToList( Global:StateVars,
                    Global:Varname );
                 };
             If ( x #= I )
              Then {
                 MakeClass( Global:Varname, Inputs );
                 AppendToList( Global:Inputs, Global:Varname );
                 };
             If ( x #= O )
              Then {
                 MakeClass( Global:Varname, Outputs );
                 AppendToList( Global:Outputs, Global:Varname );
                 };
             If ( Global:Sort #= Set )
              Then xcreate_set( Global:Varname, x );
             If ( Global:Sort #= Tuple )
              Then xcreate_tuple( Global:Varname,
                   x );
             If ( Global:Sort #= Function )
              Then xcreate_func( Global:Varname, x );
             If ( Global:Sort #= Sequence )
              Then xcreate_seq( Global:Varname, x );
             If ( Global:Sort #= Bag )
              Then xcreate_bag( Global:Varname, x );
             };
          };
      };
  });

    /************************************
     **** FUNCTION: xextract41
     ************************************/
MakeFunction( xextract41, [Set Seq],
{
If ( Seq:sort != 41 And xconstrvioln?( 41, Seq ) )
  Then PostError( "zextract - " # Seq # " is not a sequence." )
  Else If Not ( Set:sort == 21 And Member?( Global:Numerics, Set:Atype ) )
     Then PostError( "zextract - Illegal argument." )
    Else {
       Global:TempNum += 1;
       Let [ temp Temp # Global:TempNum ]
          {
          xmake_temp( 22, temp, Seq );
          SetValue( temp:sort, 41 );
          zequate( temp, zsquash( zdom_res( Set, Seq ) ) );
          temp;
          };
       };
} );

    /*************************************
     **** FUNCTION: xfilter41
     *************************************/
MakeFunction( xfilter41, [Seq Set],
{
If ( Seq:sort != 41 And xconstrvioln?( 41, Seq ) )
   Then PostError( "zfilter - " # Seq # " is not a sequence." )
   Else If Not ( Set:sort == 21 And
                 ( Set:Atype #= Seq:Btype Or
                 ( Member?( Global:Numerics, Set:Atype ) And
                 ( Member?( Global:Numerics, Seq:Btype ) ) ) ) )
      Then PostError( "zfilter - Illegal argument." )
      Else {
         Global:TempNum += 1;
         Let [ temp Temp # Global:TempNum ]
            {
            xmake_temp( 22, temp, Seq );
            SetValue( temp:sort, 41 );
            zequate( temp, zsquash( zran_res( Seq, Set ) ) );
            temp;
            };
         };
} );

    /*************************************
     **** FUNCTION: xfront41
     *************************************/
MakeFunction( xfront41, [Seq],
   {
   If ( Seq:sort != 41 And xconstrvioln?( 41, Seq ) )
     Then PostError( "zfront - " # Seq # " is not a sequence." );
   Global:TempNum += 1;
   Let [temp Temp # Global:TempNum]
      {
      xmake_temp( 22, temp, Seq );
      SetValue( temp:sort, 41 );
      For count From 1 To ( LengthList( Seq:Aelems )
                    - 1 )
           Do {
          Let [pos GetElemPos( Seq:Aelems, count )]
             {
             AppendToList( temp:Aelems, count );
             AppendToList( temp:Belems, GetNthElem( Seq:Belems,
                              pos ) );
             };
          };
    temp;
    };
  } );

    /************************************
     **** FUNCTION: xhead41
     ************************************/
MakeFunction( xhead41, [Seq],
{
If ( Seq:sort != 41 And xconstrvioln?( 41, Seq ) )
  Then PostError( "zhead - " # Seq # " is not a sequence." );
Global:TempNum += 1;
Let [temp Temp # Global:TempNum]
  {
  xmake_temp( 10, temp, Seq );
  If ( Member?( Global:Numerics, Seq:Btype ) )
    Then {
      SetValue( temp:Atype, Z );
      SetSlotOption( temp:Aval, VALUE_TYPE, NUMBER );
      }
    Else {
      SetValue( temp:Atype, Seq:Btype );
      SetSlotOption( temp:Aval, VALUE_TYPE, TEXT );
      };
  SetValue( temp:Aval, GetNthElem( Seq:Belems,
                        GetElemPos ( Seq:Aelems, 1 ) ) );
  temp;
  };
} );

    /************************************
     **** FUNCTION: xlast41
     ************************************/
MakeFunction( xlast41, [Seq],
{
If ( Seq:sort != 41 And xconstrvioln?( 41, Seq ) )
 Then PostError( "zlast - " # Seq # " is not a sequence." );
Global:TempNum += 1;
Let [temp Temp # Global:TempNum]
  {
  xmake_temp( 10, temp, Seq );
  If ( Member?( Global:Numerics, Seq:Btype ) )
    Then {
      SetValue( temp:Atype, Z );
      SetSlotOption( temp:Aval, VALUE_TYPE, NUMBER );
      }
    Else {
      SetValue( temp:Atype, Seq:Btype );
      SetSlotOption( temp:Aval, VALUE_TYPE, TEXT );
      };
  SetValue( temp:Aval, GetNthElem( Seq:Belems,
           GetElemPos ( Seq:Aelems, LengthList ( Seq:Aelems ))));
  temp;
  };
} );

    /************************************
     **** FUNCTION: xtail41
    ************************************/
MakeFunction( xtail41, [Seq],
  {
  If ( Seq:sort != 41 And xconstrvioln?( 41, Seq ) )
   Then PostError( "ztail - " # Seq # " is not a sequence." );
  Global:TempNum += 1;
  Let [temp Temp # Global:TempNum]
    {
    xmake_temp( 22, temp, Seq );
    SetValue( temp:sort, 41 );
    For count From 2 To LengthList( Seq:Aelems )
          Do {
         Let [pos GetElemPos( Seq:Aelems, count )]
           {
           AppendToList( temp:Aelems, count - 1 );
           AppendToList( temp:Belems, GetNthElem( Seq:Belems,
                               pos ) );
           };
         };
    temp;
    };
  } );

    /************************************
    **** FUNCTION: xload_spec
    ************************************/
MakeFunction( xload_spec, [],
  {
  ResetValue( Global:FileName );
  PostInputForm( "Load File", Global:FileName, "Enter File Name ( with no extension.)" );
  PostBusy( ON, "Loading Specification. Please Wait." );
  InterpretFile( Global:FileName # .KAL );
  PostBusy( OFF );
  PostMessage( "The specification " # Global:FileName # ".KAL has been loaded." );
  } );

    /************************************
    **** FUNCTION: xmake_func71
    ************************************/
MakeFunction( xmake_func71, [func],
  {
  MakeFunction( func, [ x ],
    {
    If ( x:sort != 10 Or Not( x:Atype #= func:Atype ) )
     Then PostError( func # " type missmatch" )
     Else If Not( Member?( func:Aelems, x:Aval ) )
          Then PostError( x # " not in domain of " # func )
          Else {
             Global:TempNum += 1;
             Let [t Temp # Global:TempNum]
               {
               xmake_temp( 10, t, func );
               SetValue( t:Atype, func:Btype );
               If Member?( Global:Numerics, t:Atype )
                 Then SetSlotOption( t:Aval, VALUE_TYPE,
                       NUMBER )
                 Else SetSlotOption( t:Aval, VALUE_TYPE,
                       TEXT );
```

```
                t:Aval = GetNthElem( func:Belems,
                         GetElemPos( func:Aelems,
                           x:Aval ) );
                t;
                };
              };
    } );
  } );

    /*************************************
     **** FUNCTION: xrev41
     *************************************/
MakeFunction( xrev41, [Seq],
  {
  If ( Seq:sort != 41 And xconstrvioln?( 41, Seq ) )
   Then PostError( "zrev - " # Seq # " is not a sequence." );
  Global:TempNum += 1;
  Let [temp Temp # Global:TempNum]
    {
    xmake_temp( 22, temp, Seq );
    SetValue( temp:sort, 41 );
    For count From 0 To ( LengthList( Seq:Aelems ) - 1 ) Do {
          Let [pos GetElemPos( Seq:Aelems,
                                    LengthList( Seq:Aelems ) - count ) ]
            {
            AppendToList( temp:Aelems, count + 1 );
            AppendToList( temp:Belems, GetNthElem( Seq:Belems, pos ) );
            };
          };
    temp;
    };
  } );

    /*************************************
     **** FUNCTION: xshow_var
     *************************************/
MakeFunction( xshow_var, [],
  {
  Let [x PostMenu( "Select type of variable to display.", State,
          Input, Output, Local, "* CANCEL *" )]
    {
    If ( x #= State )
     Then xshow_state_var( )
     Else If ( x #= Input )
          Then xshow_input_var( )
          Else If ( x #= Output )
               Then xshow_output_var( )
               Else If ( x #= Local )
                    Then xshow_local_var( );
    };
  } );

    /*************************************
     **** FUNCTION: xsquash41
     *************************************/
MakeFunction( xsquash41, [Seq],
  {
  If ( xconstrvioln?( 71, Seq ) )
```

```
    Then PostError( "zsquash - " # Seq # " can not be squashed." );
   Global:TempNum += 1;
   Let [temp Temp # Global:TempNum]
      {
      xmake_temp( 22, temp, Seq );
      SetValue( temp:sort, 41 );
      For count From 1 To Max( Seq:Aelems ) Do {
          If Member?( Seq:Aelems, count )
             Then {
                AppendToList( temp:Aelems, LengthList( temp:Aelems ) + 1 );
                AppendToList( temp:Belems, GetNthElem( Seq:Belems,
                                    GetElemPos( Seq:Aelems, count )));
                };
             };
      temp;
      };
   } );

     /*************************************
      ****  FUNCTION: zconcat
      *************************************/
MakeFunction( zconcat, [Seq1 Seq2],
   {
   xvalid_var( Seq1 );
   xvalid_var( Seq2 );
   If Not( Member?( Global:VarSeqCodes, Seq1:sort )
           Or ( Member?( Global:Numerics, Seq1:Atype )
                 And Min( Seq1:Aelems ) >= 1
                 And ( Seq1:sort == 22 Or Seq1:sort == 24
                    Or Seq1:sort == 71 Or Seq1:sort == 72 ) ) )
     Then PostError( "zconcat - Illegal argument." )
     Else If ( Seq1:sort == 22 Or Seq1:sort == 41 Or Seq1:sort == 71 )
          Then xconcat41( Seq1, Seq2 )
          Else If ( Seq1:sort == 24 Or Seq1:sort == 42 Or Seq1:sort == 72 )
               Then xconcat42( Seq1, Seq2 )
               Else PostError( "zconcat - Not yet available." );
   } );

     /*************************************
      ****  FUNCTION: zfilter
      *************************************/
MakeFunction( zfilter, [Seq Set],
   {
   xvalid_var( Set );
   xvalid_var( Seq );
   If Not( Member?( Global:VarSeqCodes, Seq:sort )
           Or ( Member?( Global:Numerics, Seq:Atype )
                 And Min( Seq:Aelems ) >= 1
                 And ( Seq:sort == 22 Or Seq:sort == 24
                    Or Seq:sort == 71 Or Seq:sort == 72 ) ) )
     Then PostError( "zfilter - Illegal argument." )
     Else If ( Seq:sort == 22 Or Seq:sort == 41 Or Seq:sort == 71 )
          Then xfilter41( Seq, Set )
          Else If ( Seq:sort == 24 Or Seq:sort == 42 Or Seq:sort == 72 )
               Then xfilter42( Seq, Set )
               Else PostError( "zfilter - Not yet available." );
   } );
```

```
    /************************************
     **** FUNCTION: zfront
     ************************************/
MakeFunction( zfront, [Seq],
  {
  xvalid_var( Seq );
  If Not( Member?( Global:VarSeqCodes, Seq:sort )
          Or ( Member?( Global:Numerics, Seq:Atype )
               And Min( Seq:Aelems ) >= 1 And ( Seq:sort
                                                ==
                                                22
                                                Or
                                                Seq:sort
                                                ==
                                                24
                                                Or
                                                Seq:sort
                                                ==
                                                71
                                                Or
                                                Seq:sort
                                                ==
                                                72 ) ) )
    Then PostError( "zfront - Illegal argument." )
    Else If ( Seq:sort == 22 Or Seq:sort == 41 Or Seq:sort ==
            71 )
        Then xfront41( Seq )
        Else If ( Seq:sort == 24 Or Seq:sort == 42 Or Seq:sort
                == 72 )
            Then xfront42( Seq )
            Else PostError( "zfront - Not yet available." );
  } );
```