Prototyping Z Specifications in Extended Lisp

Richard Bramwell Hibberd

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

October 2001

Acknowledgements

I would like to record my profound thanks to Ian Morrey and Jawed Siddiqi of the School of Computing and Management Sciences at Sheffield Hallam University, who have so freely offered guidance, support, encouragement and enthusiasm throughout the course of the project.

I would also like to acknowledge the work of Graham Buckberry, the creator of the companion CASE tool TranZit, which work is referenced in this thesis.

I must also record grateful thanks to successive Heads of the Department of Computing at The Nottingham Trent University, namely Professor Bob Whitrow, Dr. Ted Ashworth, Ms Pauline Fazackerley and Professor Adrian Hopgood; they all demonstrated faith in the eventual outcome, despite the extended timescale and numerous contra-indications.

It is impossible to acknowledge individually all those who have been obliged to live with the consequences of my original decision to embark on this undertaking; this includes my work colleagues and my family, but much the greatest contribution has been from my wife Janice Hibberd who has unfailingly supported me in this undertaking. I cannot see that the debt I owe her can ever be repaid, but I am glad that I now have the chance to try.

Richard Hibberd BEd MSc October 2001

For Rosie and Fraser

Table of Contents

Table of Figures

Abstract

Much research has identified shortcomings in the Requirements Description to be the key factor in the failure of many software development projects; the development of *formal specification* techniques and notations allows the unambiguous statement of requirements, against which an implementation can generally be verified or even proved. While this approach will resolve many of the difficulties, it is impossible to formally confirm that such a specification is correct with respect to the *intention* of the customer; the abstraction that is characteristic of such languages can make the formal specification inaccessible without specialist skills.

Z is one such, model-based, specification notation and this thesis reports on a CASE tool, the Z Animator in Lisp, that supports a process of specification validation through animation. A specification in the proprietary ZAL format, a high-level, largely functional, executable notation based on extended Lisp, can be executed by the *Animation Engine* within the ZAL animation environment. Using a graphical environment running under Microsoft Windows™, schemas representing the operations upon the state are animated by populating their inputs, evaluating their predicates and reporting the outcomes to the user; these outcomes can be used to directly update the state, prior to further executions. The animator ensures the consistency of the on-going model state by the execution of the system invariant. The user can identify precisely which elements of the state should be displayed and can thereby focus on the particular areas of interest. This interaction is significantly more accessible to the customer and can be used to explore properties of the specification and thereby confirm, or not, that it exhibits the desired behaviour. This process *validates* the specification with respect to *customer intention.*

Because the transformation into the proprietary ZAL language can be largely automated, using a companion CASE tool called TranZit, the process supports the

iterative development of an improved specification, since at each stage the Z document reflects the system being animated.

1. Introduction and Problem Definition

During the twenty four hours prior to writing this particular sentence, the author has interacted with software in many ways: explicitly using an Internet browser to order groceries for home delivery and to pay for these; to bid for and buy a birthday gift; to listen to an audio signal streamed several thousand miles; and to view news reports from yet further away. Other software has been used to create, to evaluate and to disseminate learning materials to students and colleagues and to communicate with a number of them. Implicitly, embedded software has controlled the operation of many of the devices used daily as a matter of course: the mobile phone with its text messaging capability; the motor car with electronic engine management and indeed software control of almost every system from air-bag deployment to traction control; the excellent central heating controller[1] that is a plug-in replacement for an older electro-mechanical device and which provides such a natural and flexible interface; ignoring the availability of 32-bit games consoles for entertainment, even forty year old slot-car racing systems now have retro-fit pacing systems that will "learn" the controller position (and consequently the voltage) for a fast lap and will provide automated competition for the solo driver.

The lesson of this arbitrary and individual review is the observation that software pervades the life of anyone living in the United Kingdom in the early twenty-first century. Indeed, its impact can be somewhat more significant than 'pervading' as is evidenced by the fatal effect of software in such cases as the Airbus 320 accident at Warsaw, Poland on 14 September 1993, in which a passenger and a pilot lost their lives; though the investigation concluded that the pilots had reacted inappropriately to data available to them (principally regarding excessive tail wind), they were not able to

[1] Horstmann Model H27

apply reverse thrust early enough to slow the aircraft sufficiently to stop within the available space. This was a consequence of a software lockout which required compression of both undercarriage shock-absorbers and which could not be over-ridden [Main Commission Aircraft Accident Investigation 1994].

And the impact is likely to increase : [Gibbs 1994] quotes Remi H. Bourgonjon, director of software technology at Philips Research Laboratory in Eindhoven as saying, ``The amount of code in most consumer products is doubling every two years."

Given that software is fundamental to so many activities and that it is a product of human endeavour, that it sometimes fails to perform appropriately is unsurprising; this failure may be inconsequential, perhaps necessitating a re-boot of a personal computer or expensive and spectacular, as in the case of the $500 million self-destruction of the Araine-5, but the lack of surprise should not be taken as conferring acceptability on such failures.

The need to minimise the occurrence of, and to ameliorate the impact of such software failures is the motivation behind the development of methods, techniques and processes to improve the quality – however measured – of software systems – these together constitute the discipline that is known as Software Engineering and this document reports on an attempt to improve the practice in one small area of that discipline, namely Requirements Validation.

1.1 The Software Crisis and palliative measures – Software Engineering as a discipline

The recognition of what became known as the Software Crisis – the consistent, almost inevitable tendency of software projects to over-run development schedules and budgets – led to much research into the causes and remedies. A succession of techniques was developed that addressed, but did not ultimately resolve, the difficulties; these include Structured Programming, the formalisation of the development process into numerous Software Lifecycle models and Object Oriented Programming. [DeMarco and Lister 1989] found that real benefits do accrue from these techniques, but still the problems of late, over-budget, inappropriate software persist. [Brooks 1987] postulates two types of software development difficulty – the essential and the accidental. An accidental difficulty might be an insufficiently fast hardware resource and these can normally be readily dealt with; what then remain are essential difficulties in that they derive from the essence of the process of software development or indeed from the essence of software. It is a highly complex, abstract activity (or product) that is not easily reduced to a series of simple steps.

In many ways the chronology of software development is the evolution of an activity from a "black art" comprising skills that seem to "work" insofar as they give rise to acceptable products, into a mature discipline with formal processes and methods and a much better defined notion of what constitutes best practice in any given problem domain; interestingly, this evolution is analogous to the process maturity classification proposed by the Software Engineering Institute's Capability Maturity Model.

The software development practices of early programming undertakings, perhaps characterised by individuals or small teams using at best the early high-level languages, did not scale at all well ; the complexity of the systems being developed increased hugely and teams of developers, rather than individuals, became the norm. To resolve the problems that flowed from this – effectively those of any nascent technology - it was thought appropriate to borrow from disciplines where these problems had already been addressed and to a great extent solved. The obvious candidate was engineering, and so was born Software Engineering, though this still reflects more an aspiration than a description. However, engineering is concerned with putting scientific knowledge to practical uses and from this perspective there is considerable doubt as to whether software engineering, as often practised, is in any sense engineering because there is little if any scientific knowledge that is put to use.

1.2 Formal Methods as a part of Software Engineering

This begs the question as to what scientific knowledge is of relevance to software engineers. One approach that attempts to provide both theoretical underpinnings and scientific rigour is the use of formal methods. The aim of formal methods is to establish a mathematical foundation for software development in two crucial and related areas where precision and rigour are of the utmost importance: specification and verification. This is achieved through the provision of a framework for the development of provably correct programs (i.e. programs, which will not deviate from their intended behaviour when executed). Its proponents claim that this approach has several advantages, some of which include: the increased confidence in the reliability of the parts which have had their correctness established; freedom from the inadequacies and limitations of testing; and perhaps most significantly, the diminishing need for corrective maintenance.

Much as the software engineers seek to adopt the methods and rigour of engineering, so the advocates of formal methods draw on the philosophy of mathematics, pursuing a deductive approach in which emphasis is placed on representing relationships in the problem domain through formal modelling and legitimating it through rigorous proof.

An alternative stems from a scientific tradition of empiricism which involves observation, hypothesis testing and theory building. Within the discipline of software engineering practitioners have advocated both traditions with varying degrees of emphasis. The proponents of a formal methods approach have tended to emphasise the need to strengthen practitioner skills in the deductive direction [Wing 1990], whilst those advocating a strong science/engineering tradition have argued that a purely deductive approach is not practical for many software systems.

The larger project within which this particular research is located combines these two traditions to form an eclectic approach that is suitable for some problems; it involves refining the given set of requirements into a formal specification building on the deductive tradition of formal modelling. The validation of the problem statement is carried out by exploring the specification in the inductive tradition through its execution.

1.3 Requirements Engineering – its place in Software Engineering

Any representation of the Software Lifecycle will include, indeed will almost certainly begin with, a phase entitled Requirements Engineering; the significance of this activity can hardly be overstated, as so many case studies of "failed" software projects have concluded that the problems began with a poor understanding of user requirements. These would include the Olympic Information Integration system developed by IBM for the 1986 Atlanta Olympic Games, whose project manager conceded *"user requirements were not understood"* [Forberg and Mooz 1997] and the London Ambulance Service computer-aided dispatch system, though poorly understood requirements is but one of many shortcomings in that project [Page *et al* 1993]. In fact any report of software project over-run is generally accompanied by a recitation of the high profile failures, much as in Section 1.1, but many analyses of the causes of the problems have highlighted errors in the capture and description of requirements as both being both wide-spread and having disproportionately significant impact.

For example, [Basili and Perricone 1984], in an empirical investigation of software errors, report that 48% of the faults observed in a medium-scale software project were "attributed to incorrect or misinterpreted functional specifications or requirements"; similarly [Perry and Steig 1993] conclude that 20.4% of the implementation faults are due to incomplete or omitted requirements.

The need to manage the requirements analysis and capture has led to many worthwhile developments both in processes and in the tools to support them; this research is intended to contribute in some way to continuing that progress.

1.4 Research objectives

This work forms a part of a larger project, ProToRE, based at the Computing Research Centre at Sheffield Hallam University; ProToRE[2] involves:

> *"the provision of Requirements Engineering technology (i.e. processes methods and tools) that assist in representing, validating and evolving requirements so as to deliver a high quality requirements document represented in forms that are appropriate to the needs of both users and developers"*
>
> [ProToRE 2002]

Within this broad aim, and indeed the title, can be identified the distinct strands that form the framework within which this work is located:

- the processes and methods are essentially those that constitute the REALiZE[3] method, which is described in Section 2.4 and
- the tools are the software components that support the REALiZE method and comprise
    - TranZit, a full-screen editor for constructing and syntax analysing Z specifications, with its associated transformation engine;
    - ZAL, a LISP-based environment for constructing an executable version of a Z specification – the subject of this report; and
    - ViZ, an object based system for visualising executions.

[2] Processes and Tools for Requirements Engineering
[3] Requirements Engineering by Animating LISP incorporating Z Extensions

ViZ represents a relatively recent area of investigation, whose development has complemented the toolset and which utilises the functionality of the other two tools, but which has impacted neither on the development of the other two components, nor on the relationship between them, nor on the way that they are used. Consequently, while TranZit will be described and discussed, and its relationship to ZAL will be examined, ViZ is considered to be outside the scope of this report.

The broad project aim of supporting the development of high quality requirements documentation is generic, insofar as the form of that documentation is unspecified; nonetheless, throughout its life the project has sought to support the development of a high quality, formal specification written in the Z notation.

The precision and absence of ambiguity of such a specification confer significant advantages in the implementation phase of a software development, but the danger exists that these benefits are at the expense of incorrectly captured requirements; the question of the quality of such a requirements document may reduce to one of how well the formal requirements match the actual requirements of the client. As a consequence, the project has devised techniques to support and guide the development of the formal specification and seeks to develop tools to validate that specification *with respect to client intention.*

From this more concrete objective has come the motivation to provide software components that will support the *creation* of a Z specification (using the TranZit tool) and its subsequent *animation* (using the ZAL tool). The primary aim would be to increase the accessibility of the specification and thereby allow its validation by the client, but such an animator would also enhance the specification *creation* process, by allowing the developer of the specification, i.e. its writer, to verify its correctness.

To investigate the form of this animation and validation and how it might be achieved is the purpose of this research.

From this can be derived the supplementary questions that form the basis of the enquiry:

- how might the process of specification validation be improved by animation?
- what form might such an animation system take and how would animation be incorporated into the requirements formalisation process?
- which aspects of the process might animation help?
- can animation facilities be provided that are sufficiently usable – i.e. that will enhance rather than simply complicate the process?
- can the inductive style of specification development be supported, in contrast to the more usual *post hoc* deductive proof offered by other tools?
- what coverage of the Z specification language might such an animation system provide and how would the system be developed?

Almost incidentally, these questions can also be seen as leading to an alternative perspective on the development of the animator, which sees the objective as seeking to use animation to enhance the accessibility of the formal notation; this viewpoint lacks the framework of formal Software/Requirements Engineering activities, but does allow for the use of the animator in a non-prescribed way, where it would be for the users to determine where and how it might be used. This perspective is entirely consistent with that of the animation as an element of the REALiZE process.

These questions have indeed been investigated and this document reports on that investigation.

1.5 Thesis structure

The remainder of this thesis reports on the research, design, development and evaluation of the Z Animator in Lisp, or ZAL, proposed in Section 1.4, the Research objectives ; also considered in some detail is a process that integrates the use of the animation system in a more formal requirements capture and validation context which has been named the REALiZE process, and which provides the context for the ZAL animator.

Chapter 2 identifies the context of this development and proposed use, by considering the broad landscape of Requirements Engineering, identifying a role for formal methods within that landscape and suggesting how the use of animation might enhance that role.

Chapter 3 considers the benefits of animation and explores different approaches to it; the place of ZAL in the REALiZE method is explained.

Chapter 4 considers the realisation of the ZAL toolset component; the underlying design is explored and its implementation considered in some detail.

Chapter 5 considers and demonstrates the use of ZAL in three separate and contrasting situations: in a conventional data processing scenario; in a safety-critical, process control situation; and as an interactive tool, supporting Software Engineering education.

In Chapter 6, the results of the research programme are assessed and conclusions drawn against the objectives identified in Section 1.4.

2 Requirements Engineering

This section will review the development of Requirements Engineering as a discipline and examine what currently constitutes good practice. It will consider the role that formal methods have in Requirements Engineering and identify some of the benefits and drawbacks of their use. A need is identified for improvement in the specification validation process and a mechanism proposed and described that will allow the integration of specification capture and validation into an iterative cycle, leading to the development of an improved formal specification with wider ownership among the stakeholder team.

The importance of this activity was identified by [Brooks 1987] in his seminal paper "No Silver Bullet – Accident and Essence of Software Engineering", uncompromisingly asserting

> *"The hardest single part of building a system is deciding what to build. No other part is as difficult as establishing the detailed technical requirements..... No other part of the work so cripples the resulting system if done wrong. No other part is so difficult to rectify later"*

2.1 *Requirements Engineering –the fundamental activities*

Requirements Engineering as a discipline might be considered to have come of age in 1993 which saw the 1st IEEE International Symposium on Requirements Engineering, though obviously practitioners had been working in the area prior to that event but without the convenience of an umbrella title and the recognition of the IEEE; references to the phrase "Requirements Engineering" can be found as early as 1979 [Alford and Lawson 1979]. The profile of those activities that now comprise Requirements Engineering had been increasing with the wider recognition that failure to capture adequately user requirements was perhaps the single most significant cause of project failure, or at least of what have been described somewhat euphemistically as projects being "challenged". One benefit of this increasing interest was that a consensus developed as to

- what the term Requirements Engineering describes and
- the broad categorisation of what activities that might be undertaken in the name of Requirements Engineering.

The problem with a consensus is that everyone understands what it means, but no two individuals will agree on the precise definition of that understanding; notwithstanding this, Requirements Engineering can be thought of as "the activities involved in the discovering, documenting, validating and maintaining a set of requirements for a computer-based system".

Following straightforwardly from this definition are the three principal phases:

- requirements elicitation
- requirements formalisation
- requirements validation.

The maintenance of the set of requirements is necessary because this is essentially an iterative process, where phases are revisited; the process of validation may highlight either an inconsistency or an ambiguity in the formal statement of the requirements, which would require further investigation and formalisation. In "The Perfect Requirement Myth", [Mullery 1996] contends

> *"In reality, on the major systems which are so notorious for disastrous failure, an initial near-perfect requirement specification exercise, followed by a minor maintenance activity is a myth which retains credibility through the inability of the development community to recognise that there in no such thing";*

this would seem to undermine the deductive approach of proving the requirements correct and consistent, given the fallibility that Mullery asserts. Regardless though of the achievability of the perfect specification, it is incumbent on those writing specifications that they write the best possible one and to this end practitioners of Requirements Engineering continue to strive.

None of the above identifies precisely what a requirement might be; a working definition could be " a requirement is a feature of a system or facility that **must** be provided in order to fulfil a system's purpose". The collection of all the requirements form the requirements definition; a number of authors [Pfleeger 1998] [Somerville 1989] differentiate this from the requirements specification, with the latter being the formal, technical document and this distinction will be observed here. It also implies the evolution of the formal statement of requirements from the requirements analysis process.

The three phases of the overall process can be examined, to locate more precisely the place of this research in the Requirements Engineering framework.

2.2 Requirements elicitation

Requirements elicitation is the process by which the requirements analyst discovers, structures, collates and records “what the customer wants”; as a result, the analyst becomes sufficiently expert in the domain of the problem to synthesise a formal statement of requirements.

There are many techniques by which this can achieved; Robertson and Robertson [Pfleeger 1998] report the Volere requirements process model, see Figure 2-1.

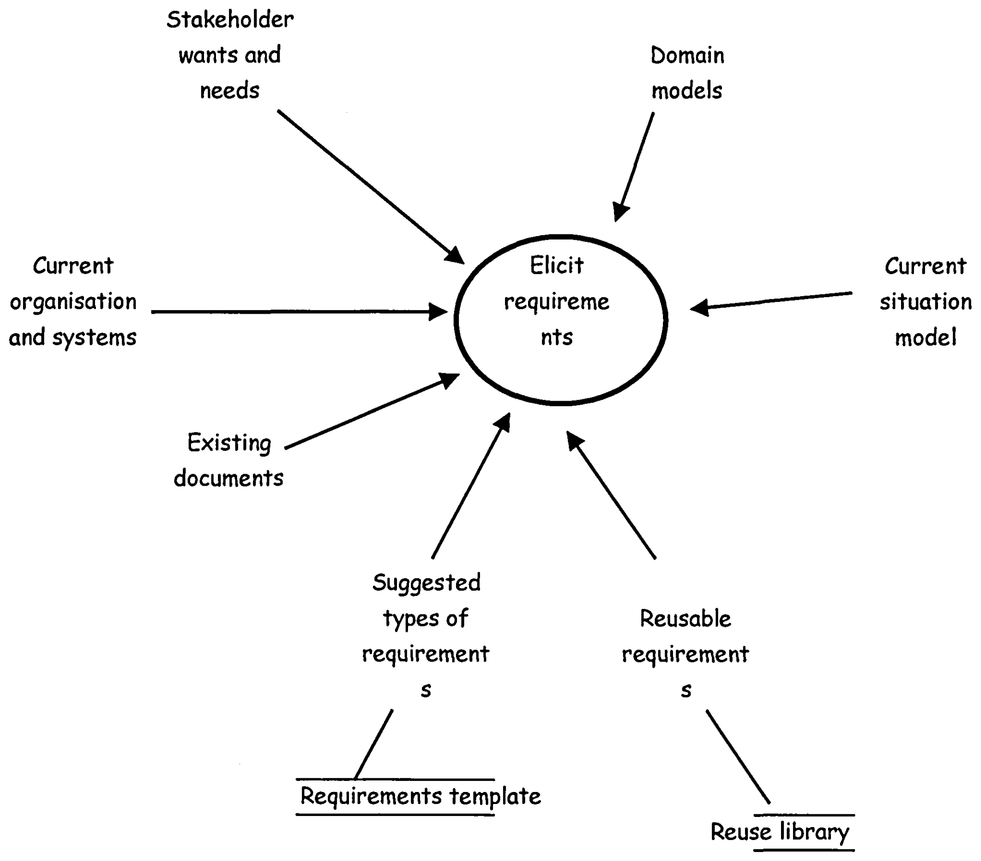


Figure 2-1 : Possible sources of requirements

For each of these possible sources, there are a number of techniques that can be applied; Van Vliet [van Vliet 2000] identifies a large number of such techniques and categorises their strengths. The table is reproduced as Figure 2-2 below:

Technique	Main information source		Strong on	
	Domain	User	Current	Future
Interview		X	X	
Delphi technique		X	X	
Brainstorming session		X		X
Task analysis		X	X	
Scenario (use case) analysis		X	X	X
Ethnography	X		X	
Form analysis	X		X	
Analysis of natural language descriptions	X		X	
Synthesis of reqs from an existing system	X		X	
Domain analysis	X		X	
Use of reference models	X		X	
Business Process Redesign	X		X	X
Prototyping		X		X

Figure 2-2 : A sample of requirements elicitation techniques (from [van Vliet 2000])

Descriptions of these techniques can be found in [van Vliet 2000] and indeed in almost every Requirements Engineering book; what is described in this report is intended to augment this collection. The animation system must have a specification to animate and so cannot be used for the first iteration, but subsequent passes can use the feedback from stakeholders when exposed to the animation to inform the process. In much the same way that a prototype cannot be the starting point for the elicitation process, neither can an animation; the animation however can feed back into the loop at a much earlier stage because it is produced largely automatically from the specification, whereas a prototype requires a significant coding effort, even with prototyping languages.

2.2.1 Requirements formalisation

Once the requirements analyst has a suitably sophisticated model of the problem domain, the requirements can be formalised into a requirements specification. This document must communicate the results of the analysis to all the stakeholders and there are a number of criteria it must meet [IEEE 1993], which include:

- it should be understandable to all the stakeholders, with whom resides its ownership, if not its authorship;
- it should be correct;
- it should be unambiguous;
- it should be consistent;
- it should be complete.

Other attributes of less significance here are that it should be traceable, verifiable and modifiable. These demands are those that proponents of formal specification techniques claim to address, with the exception of the first; it is that first requirement, for the specification document to be accessible to all the stakeholders, that the ZAL animation system can claim to satisfy.

2.2.2 Requirements validation

Our rather over-simplified model indicates that once the requirements analyst has formalised the requirements into a specification, that specification must be validated; it must be established by some mechanism that the requirements as represented by the specification correspond to the requirements as understood by the project sponsor or client (though this role or actor is often referred to as "the user"). In fact requirements reviews will generally have occurred while the requirements definition is formulated. However it remains appropriate to consider the activity as a discrete one, regardless of whether it is undertaken alongside or after the discovery and formalisation activities.

However "demonstrating that a set of requirements meets a user's needs is extremely difficult" [Somerville 1989]; [van Vliet 2000] confirms this:

> *"A major stumbling block to this stage is ensuring the user understands the contents of the requirements specification."*

and continues

> *"The techniques applied at this stage often resolve into a translation of the requirements into a form palatable to user inspection: natural-language paraphrasing, the discussion of possible usage scenarios, prototyping, and animation."*

[Meyer 1985] suggests a similar approach

> *"first describe and analyse the problem using some formal notation and then translate it back into natural language. The natural language description thus obtained will in general represent a more precise notion of the problem. And it is readable to (sic) the user. Obviously both these models must now be kept up-to-date."*

Attractive though this suggestion is, the final point is quite significant, for there is a price to be paid either to maintain the correspondence or, though more difficult to quantify, if it is ignored.

2.3 Formal Methods in Requirements Engineering

Though their proponents have never claimed them to be, Formal Methods have regularly been dismissed as "not the silver bullet", in acknowledgement of [Brooks 1987]. This is in fact no criticism at all, if one accepts the validity of Brooks' claim – if there is no silver bullet, it is tautological to then itemise anything particular that is not. These somewhat trite dismissals do however seek to make a valid point: that whatever advantages are offered by their use, formal methods are by no means a panacea for the problems of software development, but simply another technique whose use may be beneficial in some circumstances.

The exact same perceptions of failure in the software development process led some to postulate that the underlying cause was the lack of rigour in that process; this could be addressed by adopting the formality of mathematics, with formality a synonym for rigour. Deriving from this deductive tradition, the perception was of programs as mathematical entities that were amenable to processes such as proof and transformation and where the precision of mathematics would preclude any ambiguity.

A definition of Formal Methods can be found in [Leveson1990]:

> *"all applications of (primarily) discrete mathematics to software engineering problems".*

These methods divide broadly into two categories, Verified Design and Formal Specification [Jones 1990]; initial research tended to focus on the former, whereby an implementation was verified against its initial design specification by a number of *proofs*. Unsurprisingly in the light of the significant expense of this approach, it is largely restricted to the development of safety-critical systems or rather those with high potential cost of failure, in human or financial terms.

More recent effort has been devoted to production of the specification in a formal notation that can be reasoned with, both formally and informally. Within this second strand are two contrasting approaches: the model-based and the algebraic specification notations.

A model-based approach uses the structural elements of discrete mathematics, such as sets, relations and functions, and constructs a model of the system by defining the components of the state and the various operations performed upon them (and consequently is sometimes referred to as an 'operational' approach); Z [Abrial 1980][Spivey 1992] and VDM [Jones 1990] are both model-based specification notations.

An algebraic specification would be familiar to a student of functional programming, since a 'definitional' (this is in contrast to 'operational') view is taken; the definition, of generally some abstract data type, comprises a signature -the declarations of the members of the type- and an axiomatic part, where are found the re-write equations defining the rules, or axioms.

This example of a specification of the natural numbers is from [van Vliet 2000];

```
type nat
functions
   Null:           → Nat
   Succ:Nat        → Nat
   Add:Nat * Nat   → Nat
axioms
   Add(i, Null) = i
   Add(i, Succ(j)) = Succ(Add(i,j))
```

This approach is highly mathematical and no more abstract than current functional programming languages such as Haskell [PeytonJones and Hughes 1998]; though the same criticism of inaccessibility can be legitimately made of an algebraic specification and of –say- a Z specification, the model-based abstraction of the latter, with its characteristic on-going state, is much closer to an intuitive, human view of the system it models.

2.3.1 The benefits of formal specification

A formal specification is expressed in a notation whose syntax and semantics are formally defined, which will preclude the use of natural language. The major benefits of using a formal language are tabulated by [Somerville 1989], and are paraphrased here, as:

- providing insights into the requirements;
- the possibility of animation or of prototyping;
- the ability to prove the conformance of an implementation;
- automatic processing, often using software tools;
- amenable to mathematical analysis;
- providing guidance for the design of testing

and to which [Barden *et al* 1994] adds

- project management visibility.

While these are of varying relevance in the area of Requirements Engineering, they remain powerful arguments for the wider adoption of formal techniques; even with such benefits apparently readily available, there remains little evidence of their increased use.

2.3.2 Specification using Z

The reader of a formal specification will encounter a highly abstract model of the proposed system, which will abound with mathematical formulae; in order to assimilate the detail of that model easily the reader must have some expectation of the organisation of the specification, a template to populate with the details of this particular instance. One of the strengths of Z is that the basic structuring element is the schema, which emerges as highly flexible in use and which naturally supports both structural and functional decomposition. The modularity provided by using schemas allows both a top-down decomposition, perhaps more useful for an experienced reader, and yet bottom-up accessibility to the detail of declarations and operations. The schema calculus provides both for higher-order manipulation of the schemas through, for example, composition and also for the essentially straightforward textual expansion of schema inclusion. And as a bonus, it utilises graphical highlighting to enhance readability, which should not be deprecated; the consistent 'shape' of Z specifications makes for a less steep learning curve for the novice reader. No little experience of introducing formal specification in Z to undergraduates with often limited mathematical experience, using the schema as the basic structural element, has convinced the author of the value of a bottom-up approach based firmly on the schema, with its calculus a natural development of the ways it is used.

It is not proposed to provide a tutorial on the Z notation here; there are numerous books which do that very well, including [Wordsworth 1992][Barden *et al* 1994][Potter *et al* 1991][Lightfoot 1991].

It is now possible to consider the question of validation of the formal specification, which is at the heart of this work; how can both the requirements analyst and project sponsor be confident that the specification document is a complete and correct representation of the user's requirements?

2.3.2.1 Requirements validation through proof

As will be further demonstrated in Section 3.2, the role of proof in the validation of requirements is limited; proof can only establish properties such as coverage and consistency between a specification and an implementation, which are the concerns of the deductive practitioner, caricatured by a desire to "build the product right". The higher priority, at least at this stage of Requirements Engineering, is to "build the right product"; since it is impossible to validate a specification with respect to user intention by proof, an alternative strategy must be adopted.

2.3.2.2 Requirements validation through animation

The validation of a formal specification is essential to the development process; without validation there is no way to establish whether any of the 'downstream' activities –those of implementation, testing etc.- is appropriate. The specification will form the basis of all the development that follows and is the definitive arbiter in the clarification of any confusion. The fact that the specification is formal may even exacerbate any incorrect requirements that may be present; they may misrepresent the user's intention, but with regard to the self-consistency of the specification, they are not 'incorrect'. The existence of the *formal* specification may cause acceptance of what might otherwise have triggered further investigation.

The only route that will ensure confidence that the formal specification is a correct and complete representation, *with respect to sponsor intention*, is to ensure the fullest possible involvement of the stakeholders in the validation process. When the document being validated is written in a highly abstract, highly mathematical notation, this can present problems; it might be that the whole stakeholder team have experience of and confidence in reading the Z document, but if not alternatives must be explored. Extensive use of natural language comments to explain the meaning of the various elements can play a part in increasing accessibility; disciplined use of terms with clearly identified meanings may help; indeed all the techniques that have been developed to support requirements capture using informal methods can assist in the understanding of the document, but none of this addresses the fundamental flaw alluded to earlier – it is the specification that must be validated, not an explanatory comment. Any contractual obligations will be based on the formal document and it is this document itself that must be accessible to all the stakeholders.

This is the key difficulty that the techniques and tools reported upon here addresses. The toolset automates to a large extent the translation of the specification from Z notation into an executable form that can be demonstrated within an animation environment to those who are in a position to say "Yes that is the behaviour that is required".

2.4 An Introduction to the REALiZE Method

The REALiZE method (Requirements Engineering by Animating LISP incorporating Z Extensions) has been developed to formalise the interplay between requirements acquisition, requirements formalisation and requirements validation, as embodied by the TranZit and ZAL toolset. The process fits into the standard software lifecycle model at the requirements analysis phase. Following an initial requirements capture stage, the specifier enters the requirements formalisation phase, where the requirements are represented by the specifier in the Z notation using the facilities provided by the TranZit tool. Once the specifier is satisfied that the formalisation is complete, the specifier enters the requirements validation phase. Here the specifier first uses the transformation engine built into the TranZit tool, to produce an executable representation of the captured Z specification in the ZAL language. The TranZit tool then forwards this executable representation to the ZAL environment. This representation can then be executed by the specifier within the ZAL animation environment, for the purposes of demonstrating properties of the captured specification to members of the stakeholder team.

The logical relationships between the individual tool components associated with the REALiZE method are shown in Figure 2-3.

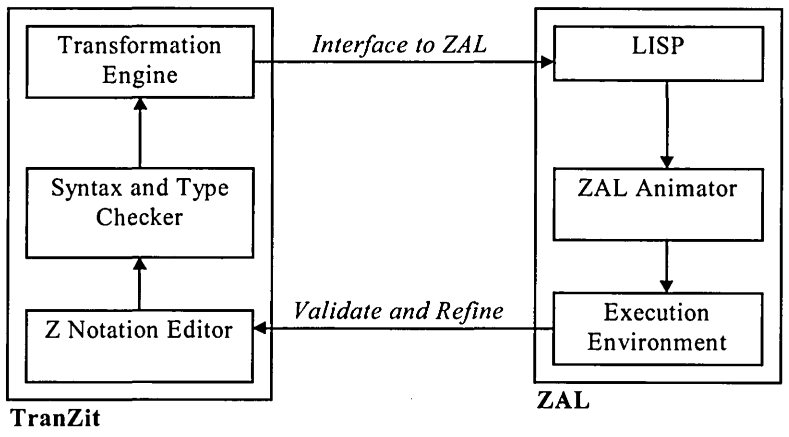


Figure 2-3 : The Logical Relationship of the Toolset Components

2.4.1 The expected benefits

The use of an animation is consistent with good practice as identified by [Somerville and Sawyer 1997]; using the taxonomy of guidelines given there, those that can be seen to validate the use of the REALiZE method are cited below:

Guideline Number	Description	How it is supported by the REALiZE method
3.1	use a standard document structure	Z is a standard notation; furthermore style guides for Z can be produced and used
3.8	make the document easy to change	tool support for this is available using TranZit
4.3	identify and consult stakeholders	the stakeholders are explicitly involved in validation by animation
4.10	prototype poorly understood requirements	the animation can be used as an automatically generated prototype
4.11	use scenarios to elicit requirements	the use described – requirements engineer sitting with the end-user in front of the prototype & walking through scenarios- is exactly that advocated by REALiZE
10.6	specify systems using formal specifications	this is self-evident, but is explained "it is important that the project customer is convinced of the value of using formal specification and that you are careful that the customer can understand the specification."

Table 1 : Requirements Engineering Good Practice (from [Somerville and Sawyer 1997])

2.4.2 Support for other software development activities

Though the work of the project is firmly located in the area of specification validation, the flexibility of the ZAL tool will allow for its use in other situations. The design of the animator has separated the user-interaction from the operation of the animation engine, as shown in Figure 2-4. This de-coupling is evidenced by the availability of a command line interface at which explicit calls to execute schemas can be evaluated.

This behaviour offers the prospect of support for other Software Engineering activities such as change control and maintenance. For a given specification, a library of test executions could be maintained; these would then be available for the automated validation of successive evolutions of the specification.

This is an area of current research effort, and this usage has not yet been formalised, but as maintainability of a specification is certainly an issue with respect to its quality, this would align with the ProToRE project objectives. Figure 5-16 demonstrates a schema execution that is instigated from a script file; this script could be seen as one such test in such a library.

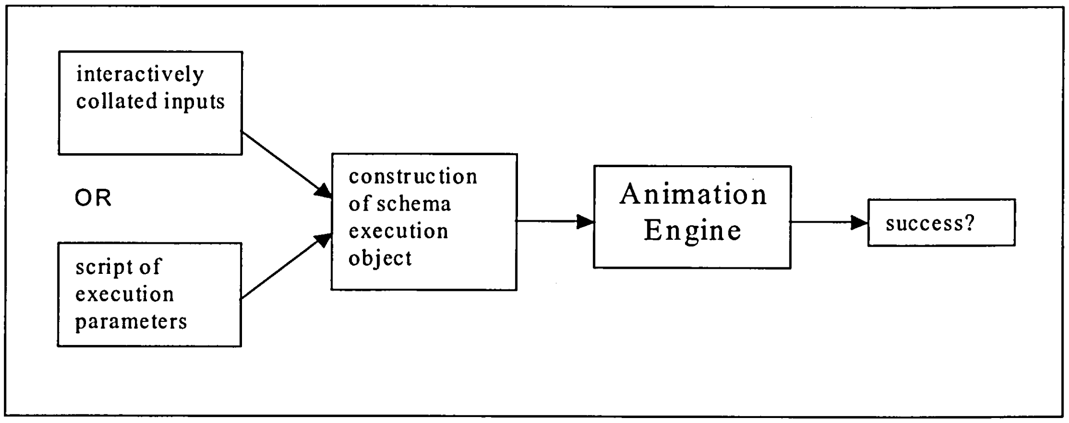

interactively collated inputs
OR
script of execution parameters
construction of schema execution object
Animation Engine
success?

Figure 2-4 : The Modularisation of Schema Execution

The context of ZAL within the REALiZE method and of that method within the wider framework of Requirements Engineering and Software Engineering has been identified; the enhancements which an animation system might offer to the Requirements Engineering process, and the key activities that would be facilitated or supported by such an animator have been detailed. It is now appropriate to consider the general and specific issues pertaining to the animation of formal specifications.

3 The Animation of Specifications

This section is concerned with the wider view of the execution or animation of specifications; historically, there has been some debate regarding the value or otherwise of this feature. That this project aligns with those supporting the idea will be no surprise, but the arguments both in favour of and against execution will be examined. It will emerge that the approach taken here and, more generally, in the REALiZE method addresses the concerns of both schools of thought and seeks to gain many of the advantages of executability without unduly compromising the expressive power available to the specifier.

There is no longer much debate about the need for better specifications and much of Requirements Engineering is concerned with the capture and validation of requirements in a formal specification; though there are some classes of software system (such as interrupt driven systems) that present serious difficulties in their formal specification which are best addressed by the language designers, equally there are other classes of problem which are well suited to the currently available set of formal specification techniques and which can be usefully specified, yielding many of the benefits detailed earlier. That this last category tends not to be formally specified is to a large extent due to a lack of familiarity with the techniques and notation of formal specification; the work reported here seeks to use animation to enhance the accessibility of the formal notation and thereby facilitate its wider use.

3.1 The disadvantages of specification executability

If we accept the desirability of wider use of formal specification, there remains some debate concerning the desirability of executing these specifications; [Hayes and Jones 1989] argue strongly that executability is not a desirable characteristic of a specification language. Starting from the reasonable position that *"a specification written in a notation that is not directly executable will contain less implementation detail than an executable one"*

they claim executability unduly constrains the specifier in a number of ways. Their principle objections can be detailed thus:

- Executable specifications tend to over-specify the problem.
- Combining clauses in a specification; except sometimes in Prolog, conjunctions cannot be formulated in programming languages. This is particularly true when one clause constrains an otherwise infinite search space. In a similar vein, negation cannot readily be used in an executable specification.
- Using quantifiers, as in *is-perfect-square(i)* $\triangleq \exists j \in \mathbb{N} \cdot i = j2$, will often present difficulties; this execution will probably terminate if *i* is indeed a perfect square, but searching for *j* when *i* is not a perfect square requires reasoning about the implicit mathematical property - the enumeration must stop when *j* reaches *i*.
- Non-computable problems; an executable specification must allow the formulation of specifications that are not computable - these are by definition not executable.
- Non-determinism is in general difficult ([Hayes and Jones 1989] claim impossible) to model; they expect that a deterministic solution is the best that could be expected.
- Some values, such as real numbers, cannot be represented; therefore a specification using reals cannot be executed, since these must be modelled using floating point approximations.
- Specification variables are used to describe non-functional requirements; how should these variables (and the requirements) be treated in an executable specification.

Significantly though, they conclude by distinguishing between specification and prototyping, and it emerges that their concerns are largely with the latter, given that

"... much of what is described in the literature as executable specifications would be better classified as rapid prototyping"

This is in fact a helpful distinction, as it serves to delineate the scope of the objections and to clarify this evaluation of the extent to which those objections are pertinent to this work.

3.1.1 The disadvantages investigated

These difficulties might seem sufficiently serious to discourage exploration of the possibilities of execution, but in fact they are refuted by both [Fuchs 1992] and [Andersen *et al* 1992] ; the approach in both cases is to provide executable examples of those problems identified by [Hayes and Jones 1989]. These counter-examples do indeed address the problems with the loss of little expressibility and the argument is made that loss of expressive power is a small price to pay for the advantage of executability. This is a slightly different analysis to that advocated here, where the unconstrained use of Z is allowed, and its animation may or may not be possible; Section 3.2.1 expands on this point.

The details of the counter-examples will not be examined, but a number of general conclusions are drawn by [Fuchs 1992].

- A general approach to transforming non-executable specifications into executable specifications involves reformulation and the addition of a small number of constructive elements, including
    - representations of sets and sequences by lists;
    - construction of sets and sequences by recursion;
    - representation of the predicate $\geq$ by generators of elements.

"Executable specifications generated in this way are direct translations of their non-executable counterparts. Since they are built from available powerful predicates they are problem-oriented, declarative and highly abstract"[5]

[5]This technique is close to that already adopted in this project.

- Many of the derived executable specifications are based on a generate-and-test approach.

- It is not enough that a specification postulates an object without detailing how the object might be constructed; the reference of [Hayes and Jones 1989] to unrepresentable values (primarily real numbers) is cited by Fuchs as an example of a specification being incomplete since it refers to a body of knowledge that is assumed by the specifier to be shared by readers of the specification. Rather than accept this as an argument against constructive specifications, they are

  "convinced that a specification, as an abstract definition of something that will have to be concretely realised, must be constructive, in the sense of constructive mathematics ..... which is intolerant of methods affirming the existence of things of some sort, without showing how to find them".

This however is a philosophical point and constitutes a counter-argument, rather than a refutation.

- Specification variables can be considered part of the (executable) specification - the animation of the specification may or may not fulfil the specified constraints, but in either case valuable information can be gained. In view of the growing proportion of current software systems that pertains to input and output, this concentration on functional behaviour is unfortunate.

It can also be argued that the constraints captured in this way refer to the ultimate implementation and not to the specification, so a failure of *the specification* to satisfy such a requirement is of no significance.

Fuchs concludes by identifying more general advantages of validation using an executable specification, namely

- the results of the validation (testing) are of much greater value because they are available much earlier in the development process;

- testing executable specifications is more *efficient* as it occurs at a more abstract level, and also in the problem domain.

While these arguments were made as a rebuttal of the [Hayes and Jones 1989] contentions, they introduce a number of the benefits of specification executability which are now examined.

3.2 The advantages of specification executability

This section establishes the potential benefits of executable specifications; consideration of how that executability might be achieved, and at what cost, is deferred until section 3.3.

[Breuer and Bowen 1994] identify the general advantages, in this case explicitly for Z :

> *"Any executable interpretation would certainly be very useful to software engineers, because it would allow Z to be used as a prototyping language as well as a specification language, and improve the interactiveness of the design process."*

The objective of the work reported here is the creation of a better *specification*, where the improved quality derives from the validation of the specification by the sponsor, among others, rather than the specifier alone, and this improved accessibility is a consequence of the executability.

The benefits of a formal specification, as detailed in 2.3.1 , can be realised only if it is a correct specification; the self-consistency of a specification, one dimension of correctness, can be established by proof, but there are other dimensions :

> *"correctness of a software system means correctness with respect to the requirements, i.e. with respect to explicit and implicit user intention and needs".*

This implies that users must be involved in the validation process;

"This suggests that the conceptual level provided by the specification is the appropriate level for the user involvement, and that validation should preferably take place in the specification phase."

since

"Executable specifications result in greater involvement by the users. Users can participate in the formulation of the specifications and in the immediate validation"

Correctness with respect to user intention is not amenable to formal reasoning; the sponsor can confirm or deny that the formal specification embodies the actual requirements, but any proof requires the existence of a formal statement of the requirements as the starting point, and it is this statement with which the reasoning is concerned. [Johnson and Sanders 1989] agree that

"Validating that a formal specification meets the customer's requirements cannot, by definition, be a formal process".

Consequently the correctness of the relationship of the specification to the actual user requirements must be established some other way; many authors identify this validation of the specification as crucial [van Vliet 2000] [Somerville and Sawyer 1997].

The meaningful evaluation of the specification as a correct statement of the requirements can only be undertaken if the specification is accessible to the sponsor; the native Z text may be readily understood by a sponsor, but that cannot be relied upon. Alternative strategies to validate the specification are needed and animation is one such approach. [Johnson and Sanders 1989] again suggest

"One technique that has some merit is to produce prototypes from formal specifications and demonstrate these prototypes to the customer".

There are other advantages. Executable specifications allow the demonstration of the behaviour of a software system before it is actually implemented, with the following benefits:

- executable components are available much earlier than in the traditional life-cycle, thereby allowing the earlier (less expensive) detection and correction of problems. This is largely a re-phrasing of the "better specification" point already addressed;
- requirements that are unclear can be clarified by animated interaction with the specification;
- execution of the specification supplements inspection and formal reasoning as a means of validation.

Furthermore, execution enables the self-consistency of a specification to be established; this facility is provided by a wide range of specification tools, by some as core functionality, but in an animation environment such as the REALiZE toolset it is a natural consequence of automatically verifying consistency by executing the state schema predicates – see Section 5.

3.2.1 How ZAL manages non-executability

As was acknowledged in section 3.1, not all Z specifications can be executed; the simple use of –say- the natural numbers, which constitute an infinite set, precludes the animation of the specification as written. As a specifier using the toolset may use whatever (legal) Z is deemed appropriate, a strategy is necessary to resolve this apparent paradox.

Buckberry details an "eclectic transformation strategy" [Buckberry 1999], which is embodied in the TranZit transformation engine and which identifies what [Breuer and Bowen 1994] term "enumeration functions" to provide candidate data for both existentially and universally quantified clauses. These clauses are transformed into ZAL expressions and consequently issues of non-computability, essentially the description of an infinite search space, have been resolved before ZAL is required to evaluate a quantified expression. The strategy involves identifying a constraint on such a search space, either explicitly from the Z expression, or implicitly where manual human intervention is employed to supply it; where the first approach fails to identify a suitable constraint, a finite subset of potentially infinite data must be instantiated by the user in order to use the animator.

It can be inferred from this that ZAL does not in fact have to consider these questions of executability and consequently is not considered further in this report.

3.2.2 The best of both worlds

At the risk of undermining the symmetry of the discussion, it can be noted that Hayes and Jones are primarily concerned that the specifier is limited in some way by the need or wish to execute the specification; this might arise either by choosing an explicitly executable notation for the specification or by using a subset of notation and techniques that will facilitate execution.

Since the REALiZE method does not constrain the specifier in the Z that can be used, the question of executability is not considered as the specification is constructed; any concessions to executability will only be made at the stage of the *transformation* of the Z into ZAL code. As the Z specification will remain the 'document of record', the issue of executability will have no impact on the specification construction process.

Neither do their concerns such as introducing an "algorithmic structure" to the specification arise, since Z has no facilities to capture algorithmic detail; the only scenario whereby the executability might constrain the specifier would be to encourage a tendency to revisit techniques and models that have proved effective in the past.

Executability is not the factor here, it is simply that experience has that effect on humans.

Furthermore, when the specifier chooses to use non-executable types or constructs, it is likely to be exactly those highly abstract details of the specification that will be least readily understood by the non-specialist; the advantage of expressiveness is potentially compromised by that expressiveness rendering the specification less accessible. In this particular situation, the availability of an animation should serve to clarify the abstract; whatever dialogue is necessary to transform the specification into an executable form cannot but ground the abstract model or mechanism in a more concrete, intermediate representation.

Notwithstanding the efforts to refute [Hayes and Jones 1989], many of the points they make are indeed valid - specifications are not *necessarily* executable; those that cannot be executed can generally be recognised as such, or constrained to make them executable – see [Buckberry 1999]. The significant subset that can be executed, either unaltered or with very little modification, are those of concern and it is our contention that advantages may generally accrue when the specification can be executed.

When a specifier writes a specification that cannot be animated, its validation must be by techniques other than execution; such a specification will remain less well validated, at least with respect to the *intention* of the sponsor.

This section concludes with a supporting quote from [Hayes and Jones 1989]:

> *"Specifications are intended for human consumption – they provide a communication link between the specifier and the user[6], and the specifier and the implementor".*

[6]The "user" is the actor referred to herein as the sponsor.

This report contends that the specifier-sponsor communication is significantly enhanced by executability and that the specifier-implementor dialogue can be unchanged, since the specification remains a Z document with no "injury" suffered in the cause of executability; it can be argued that the executability, or otherwise, of this document is of no consequence between the specifier and the implementor. However, if the specification is not accessible to the sponsor, any commitment to it cannot be considered informed.

3.3 Achieving executability

There are a number of possible categorisations for execution techniques and animation systems; some general patterns will be identified and then two frameworks will be considered in some detail to identify the appropriate context for the REALiZE method, and more particularly for the ZAL animation tool.

3.3.1 Approaches to the execution of Z specifications

In the numerous attempts to execute formal specifications, some characteristic patterns can be observed; perhaps the most significant is the way that the correspondence between the initial formal specification and the executable counterpart is established. The usual way is to take a specification and refine it (perhaps numerous times) until it is in a form that is executable, in a way analogous to the approach adopted by [Wordsworth 1992] for the implementation of a software system that has been formally specified (in Z); the emphasis is on maintaining the correctness by using techniques and transformations that are proven correct and thus establishing the correctness of the implementation. Whilst not a technique for executing specifications, this approach can be considered an extreme example of that adopted by many seeking to execute the specification; in particular [Valentine 1991] refines a subset of Z called Z-- into an increasingly concrete, less abstract form, until such time as it can be interpreted.

Broadly similar approaches have been adopted by [Sherrill and Carver 1993], who use Z as a design language for a system implemented in the functional language Haskell; by [Goodman 1993] who again uses Haskell to model Z (though the main purpose is to demonstrate the use of a monad in a functional language); by [Dick *et al* 1990] who transform Z into Prolog using correctness-preserving transformations, also known as formal program synthesis; by [West and Eaglestone 1992] who do much the same and contrast it with structure simulation, though without generators to provide candidate solutions; and by[Hörcher 1994] who has implemented a predicate compiler that transforms Z specifications (in a particular and restricted style) into C functions, which provide an exhaustive test framework. Of further interest in this last work is the original resolution (or rather avoidance) of a number of difficult execution issues by the adoption of a requirement that the user supplies expected outcomes, which are then validated; this approach is described by [Utting 2000] as a 'test oracle', and is suggested as a mechanism by which proving tools, as opposed to testing tools, can be used to validate specifications. This dichotomy is examined in section 3.3.1.2, Proving or Testing

3.3.1.1 A formal framework for classifying animators

[Breuer and Bowen 1994], in what has become a key treatment of the problem, propose the following classification of techniques for the animation of Z, based on the treatment of sets:

(a) sets must be finite and are modelled by finite arrays;

(b) sets may be countably infinite and are modelled by an enumeration algorithm;

(c) sets are cardinally unbound and modelled by the characteristic function,

though they note that class (b) and class (c) are equivalent.

This framework was designed to reflect increasing *correctness*, rather than the more usual measures of *coverage* (the portion of the grammar of Z that can be executed), *sophistication* (the termination properties of an animation) or *efficiency* (how quickly a result is obtained).

3.3.1.2 ZAL in this landscape

In this classification ZAL is currently a class (a) animator; lazy evaluation would be necessary to satisfy the class (b) criteria, which remains a possibility despite the fact that ZAL has inherited the eager evaluation of Lisp. Animation of the existential quantifier, ∃, will always need to generate candidate solutions which can then be tested in some way. An eager evaluation strategy will require all those candidates to be generated before any is tested; this is sufficient to limit this approach to class (a) status. A lazy evaluation strategy would allow much greater flexibility; in general, a value is only generated when it is needed and consequently the modelling of infinite structures (perhaps ℕ) is very much easier. For the specific problem of existential quantification, this might well provide for a more effective model, as a single satisfying value is sufficient, though failure of the quantification when no such value exists would need to be managed. Lazy evaluation is not a feature offered directly by Lisp (though the dialect Scheme [Steele and Sussman 1975] does explicitly support continuations) but [Graham 1994] suggests implementations of the facilities (macros) needed to model continuations. A continuation is a functional object that embodies the "future" of a computation, in that it is a suspended evaluation that can be called; it must "contain" all the contextual information necessary for evaluation (such as bindings that are in scope) in much the same way that as a closure must. In fact a continuation is a generalisation of a closure - a closure is a function plus pointers to the lexical variables visible at the time it was created, whereas a continuation is a function plus a pointer to the whole call stack that was pending at the time it was created. These would appear to offer a route towards a lazy evaluation mechanism, at least for the generators needed for quantifiers and comprehensions, and which might prove beneficial in other areas.

[Utting 2000] suggests and implements a possibly better, more generalised strategy in a tool called Jaza; this uses multiple (up to twelve) alternative representations of sets where

> *"Each set is kept in its optimal representation, and translated into another representation only when an operator requests it".*

This approach makes explicit the recognition that different representations are more or less well-suited to different operations.

However, fundamental changes such as these would have a significant impact not just on the animation engine, but also on the other toolset components, since potentially infinite search spaces have already been constrained, as a part of the transformation process, before the specification is presented to ZAL. Further research is indicated in this area.

3.3.1.3 Proving or Testing

As was mentioned earlier, [Utting 2000] categorises tools for analysing Z specifications depending on whether they attempt to show universal properties (Proof-like Tools) or existential properties (Testing Tools); the two groups are identified as complementary and it is stated that

> *"testing tools are better used early in the system life cycle"*

which aligns very closely with the position adopted in the REALiZE project. ZAL is clearly a testing tool and the toolset supports the very earliest stages of system development.

The use of proof has been established as inappropriate for the validation of specification *with respect to sponsor intention*; consequently it is unsurprising to note that ZAL exhibits virtually all the identified characteristics of testing tools, but the distinction is nevertheless useful as it validates the pragmatic rationale that has underpinned much of the development of the REALiZE toolset.

3.4 The REALiZE Method and its supporting toolset

The REALiZE method (Requirements Engineering by Animating LISP incorporating Z Extensions), has been developed to formalise the interplay between requirements acquisition, requirements formalisation and requirements validation, as embodied by the TranZit and ZAL toolset. The process fits into the standard software lifecycle model in the requirements analysis phase, as can be seen in Figure 3-1.

The initial Requirements Acquisition phase is entirely conventional and utilises familiar techniques such as interviewing domain specialists and user questionnaires; however the formalisation of those requirements in the Z notation is facilitated by the TranZit tool, with extensive support for creation, edit and analysis of Z documents. Once the specifier is satisfied that the formalisation is complete and that the specification captured is the best representation of the requirements possible at this stage, then the requirements can be validated. The specifier now uses a transformation engine built into the TranZit tool to produce an executable representation of the captured Z specification in the ZAL language and this executable representation is forwarded to the ZAL environment.

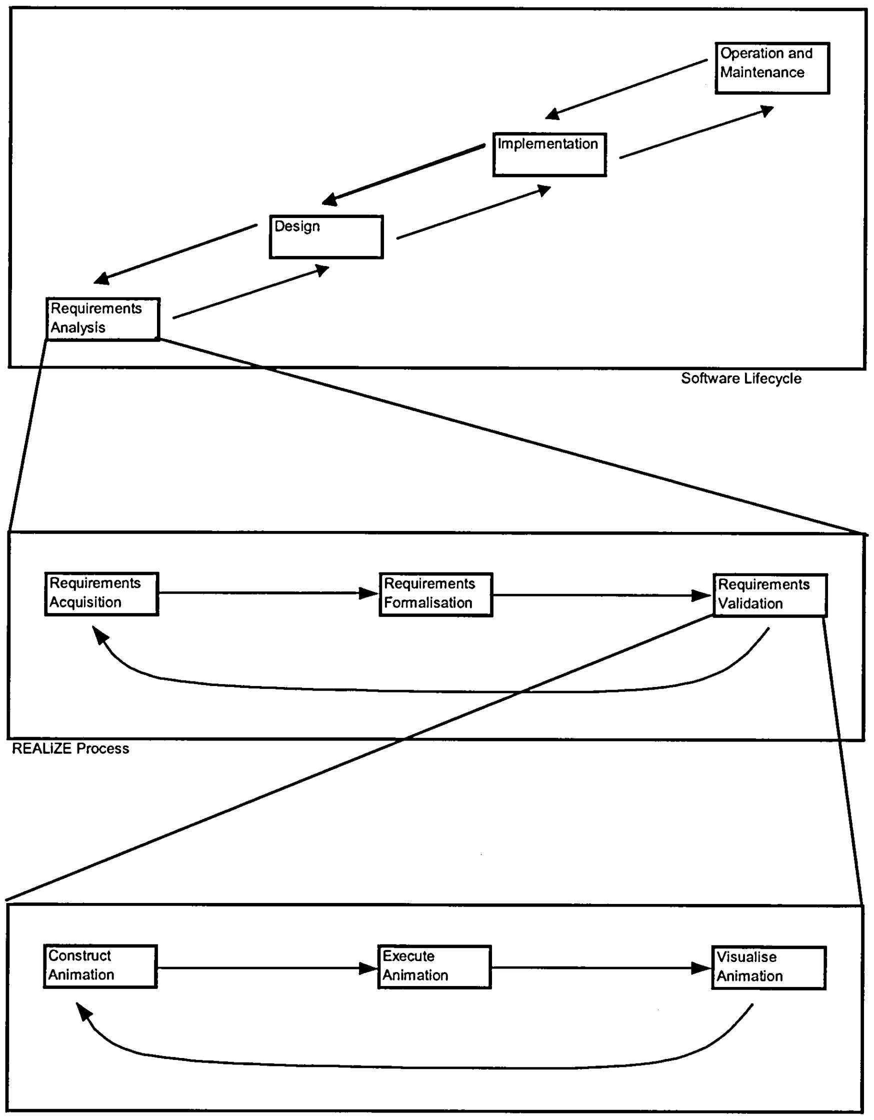


Figure 3-1 : The REALiZE method

This representation can then be executed by the specifier within the ZAL animation environment, in order:

- to confirm the correctness of the Z, with respect to the *specifier's* intention, in a way analogous to software testing;
- to demonstrate properties of the captured specification to members of the stakeholder team, using techniques such as *Scenario Walkthrough*, *Provocative Investigation* and *Exploratory Investigation*

This review is likely to lead to further iterations of the process, involving modifications to the Z document, transformation of this modified specification and further executions.

This process should enhance the understanding of the specification for all the participants and further improve the quality of the specification by ensuring that the requirements embodied are a true representation of what the system needs to do.

The logical interfaces between the individual tool components associated with the REALiZE method are shown in Figure 3-2. The reader will have noted a third tool, namely ViZ [Parry 2001], another CASE tool being developed to further enhance the interaction between the specifier and the other stakeholders by providing a visualisation of a specification. As identified in Section 1.4 and though it uses the ZAL Animation Engine to evaluate expressions and to provide results, ViZ is an entirely separate tool and will not be reported upon here.

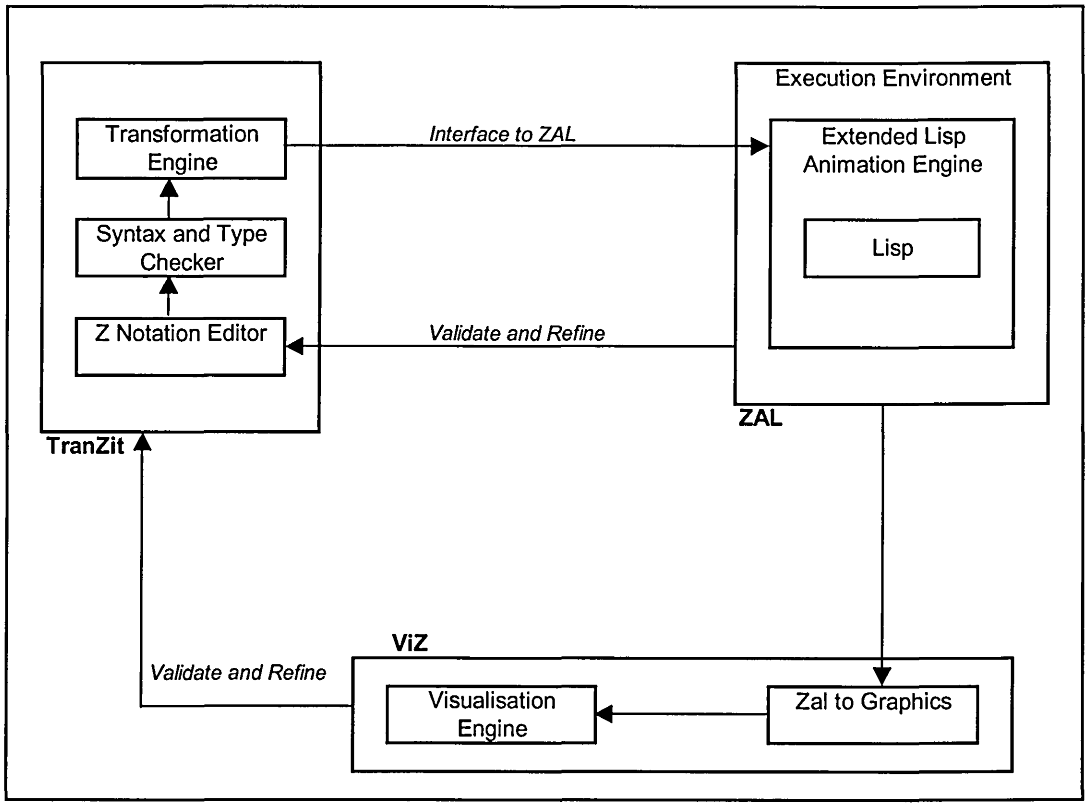


Figure 3-2 : The logical interfaces of the toolset components

3.4.1 The toolset components

TranZit (Z Editor and Transformation System)[Buckberry 1999] is a tool for capturing Z specifications, and automating their transformation to an executable representation. It incorporates features supporting the construction, manipulation and maintenance of Z specifications, as well as tools for checking their internal consistency, including a complete syntax analyser and type checker. The Z editor component of TranZit includes all the major features expected of a standard editor. In addition, full support for the Z notation character set is provided, and schema graphic outlines for standard, generic and axiomatic schemas are automatically generated. TranZit also incorporates a Transformation Engine, which allows captured specifications to be automatically transformed (as far as is possible) into an executable representation, suitable for input to the ZAL animation environment.

ZAL facilitates the exploration of Z specifications through execution, which can involve confirmation or refutation of various properties of the original specification by executing specification scenarios, thereby validating requirements. The mechanisms by which this is achieved are described in section 2.0 and demonstrated in Section 0.

3.4.2 The Edit, Transform, Execute cycle

As can be seen from Figure 3-1, the key and original activity that underpins the REALiZE process is the iterative cycle of specify, transform and execute; this is the mechanism by which successively more precise and correct (again with respect to both sponsor and specifier intention) versions of the specification are refined. Though this report is concerned with the ZAL toolset component, the REALiZE process is the context within which it works; this activity within the process should be recognised as the *raison d'être* for its development. The detailed consideration of performance of the ZAL component is undertaken in Section 5 .

The end result of this iterative process should be a better specification in which all the stakeholders have confidence .

3.4.3 De-coupling TranZit & ZAL

The REALiZE method is a formalisation of the logical interaction between the toolset components and as such supports, and to a large extent requires, the iterative development of a formal description of the requirements of a proposed system; it is the mechanism by which the integration of the two toolset components is achieved. Without ZAL, TranZit is primarily a Z editor, albeit a fully-featured one; the animation environment is needed to realise the benefits of enhanced accessibility. Without TranZit, ZAL is primarily a desktop calculator for an extended Lisp that models Z expressions and schema; the *automated* transformation is needed to guarantee the correspondence of the Z specification and the animation that is being demonstrated.

However, whilst appearing closely coupled, both TranZit and ZAL are separate, well-defined software systems, each of which can be used independently of the other; it is the REALiZE method that describes how they interact and which establishes the coherence of that interaction. Despite the references made to the TranZit tool and the REALiZE process, it is the research and development of the ZAL animation environment upon which this thesis reports.

4 Realisation of the ZAL component

Having decided to animate Z specifications, an exploratory approach was adopted, consistent with both the nature of the problem and the lack of readily-defined boundaries. This was a recognition of the ill-definition of both the task and the form and functionality of the ultimate deliverable, and to some extent of the characteristics of the host language and development environment that would best support this approach. This also reflects the inductive style adopted for the use of the toolset itself.

4.1 General issues of symbolic execution of Z

Z is a notation for specifying information systems [Spivey 1992] and was not conceived as an executable "language" in the sense of a computer language; as a consequence, there are no execution semantics associated with a specification in this notation. What has been attempted in this project is to animate such a specification, as an example of the behaviour that might be expected from an implementation of the same.

Much has been made of the problem that non-determinism poses for an animation system; Section 3 considered the objections raised by [Hayes and Jones 1989] and how these can be refuted or at least addressed. An altogether more pragmatic stance was adopted in this work that is again consistent with the fundamental choice of an inductive approach; it is that though non-determinism might appear to be crucial, a ZAL animation is always 'one possible execution', which is all that is necessary for the purpose intended.

There are two strands to the execution strategy devised and reported on here, namely the expression-level manipulation; and the encapsulation of that 'low-level' functionality into schema-level behaviour which is the principal concern of the user actor in the Requirements Validation scenario outlined in Section **3.4**. This dichotomy is manifested both in the logical design and, as a consequence, in the implementation of that design; it also serves as a useful shorthand for the conceptual distinction between the two.

Fundamental to the effective symbolic execution of Z schemas is the choice of the model of evaluation, or rather models, since again the evaluation strategy also differs in the two areas identified previously.

4.2 The application domain and symbolic evaluation strategies

[Abelson *et al* 1985] suggest that the design strategy that we choose in order to model a system is dictated by our perception of that system and suggest a number of alternative organisational strategies that could be adopted. This need to address "how a computational object can change and yet maintain its identity .... will force us to abandon our old substitution model of computation in favor *(sic)* of a more mechanistic but less theoretically tractable *environment* model of computation" [Abelson *et al* 1985] (page 168). These computational alternatives precisely correspond to the **expression level** functionality modelled using an applicative approach and to the state-managing encapsulating behaviour, or **schema-level** functionality, modelled with object-oriented techniques, though without explicit OO technologies.

That the underlying computational behaviour is applicative might well be expected – the core nature of specifications is that they are declarative, describing the *what* not the *how*, and so any attempt to model the behaviour of those specification is always likely to be declarative in style.

> *"They describe what the system must do without saying how it is to be done."*
>
> [Spivey 1992] (p1)

It is useful to consider in more detail the characteristics of the problem domain, and to examine the range of implementation strategies and vehicles that might best map onto that domain. This is the context in which design decisions with far-reaching consequences were made; the implications of those decisions cannot properly be judged until Section 5, but their validity can perhaps be established here.

4.2.1 Particular characteristics of this development

These then are the parameters that will constrain the development of the ZAL toolset component

- *the model of use in the REALiZE method*
- *the interaction with other components (principally TranZit)*
- *the problem and the programming process (incremental development)*

What are now the components of the REALiZE method, TranZit & ZAL were conceived as the CASE tools to support the logical functions that constituted a possible mechanism to better support requirements capture. This decomposition (see Figure 2-3) came to be formalised as the REALiZE method but the initial component/process interface was neither well-understood nor well-defined. As a consequence, an exploratory approach was considered appropriate to the development both of the component interactions and of the ZAL component itself.

The relationship between TranZit and ZAL is essentially that of producer and consumer, and to describe their interaction as dialogue would be to overstate its reflexivity; more properly, the TranZit tool produces a representation of the Z of interest in a form that is then executed by the ZAL tool. To have begun with a formal description of this interchange format would have enabled a more conventional development process, but the initial remit was to establish the feasibility of execution and then to investigate the range of specification techniques and styles that could be executed. In particular the parallel development of the toolset components required that this interchange format be flexible, or at least extendable to accommodate an evolving set of executable constructs and expressions. Having reached this stage of reporting on the work pre-supposes that the decisions made have to a large extent been valid, or at least correct in a sense analogous to the 'magic coin' used by [Harel 1987] in his treatment of non-determinism.

Notwithstanding quite profound reservations regarding the validity of the basic thesis argued by [Hayes and Jones 1989] (see Section **3.1.1**), they can certainly be supported in the contention that it is unquestionably no part of the role of either the specifier or the specification to address the detail of the implementation. They would want total freedom from consideration of executability, while it will be shown that minor compromise on this point can engender significant reward.

4.2.1.1 Towards a development paradigm (or 2)

The factors determining the choice of programming paradigm included, in something approaching decreasing significance :

- the characteristics of the problem domain;
- the need to integrate this tool into a larger framework;
- the experience and preferences of the developer.

The characteristics of the problem domain strongly suggested a declarative approach and no alternative was seriously entertained; this coincided with the inclinations of the developer and was considered neutral with respect to the need to integrate. As the toolset components and functionality were de-coupled from the outset, the need to integrate was explicit; it was ultimately addressed by using an extended Lisp format, where any shortfall in functionality could have been recovered by the generation of code at a lower level of abstraction (i.e. more Lisp-like), though this contingency did not arise.

The choice was resolved further, between alternative declarative approaches, again with minor deliberation. The logic languages (principally Prolog) though well suited to search-space problems, are not a natural match for the expression-based architecture of Z and [West & Eaglestone, 1992] identify a number of difficulties with using Prolog. A basis in the evaluation of expressions is of course fundamental to functional languages, such as Haskell and Lisp, and characterises the "natural fit" between problem and solution. Notwithstanding this, the functional style can be used to advantage in C++ [ISO/IEC 14882 1998], which now sports partially applied (curried) functions (using `bind1st` etc.) and higher-order functions (using `transform` and `foreach`). This question is explored in a little greater depth in Section 4.2.2

4.2.1.1.1 An illustration of a natural fit - local bindings

It may be helpful to illustrate the question of a "natural fit" by looking at the mechanism by which local bindings are established and managed in, firstly, Z and then in functional languages; this is but one of many such examples that could have been chosen. The ability to establish a local (i.e. temporary) binding for a name is useful from at least two perspectives, computational efficiency and expressive clarity.

In a computer language, by definition executable, performance benefits can accrue from only evaluating an expression once; a simple example can illustrate this -please note the careful non-avoidance of the "prototypical boring programming problem" [Harvey & Wright 1994], calculating the roots of a quadratic equation, using the following formulae:

$$r_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} \qquad r_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

The discriminant (the expression whose square root is taken) in the formulae would normally be calculated only once, 'stored' locally and looked-up when needed. This would give a C++ version of

```
std::pair<float, float> rootsOf(int a, int b, int c)
{
        float routeD ;
        float r1, r2;
        routeD = sqrt(b*b-4*a*c);
        r1 = (-b+routeD)/(2*a);
        r2 = (-b-routeD)/(2*a);
        return std::make_pair(r1, r2);
}

std::pair<float, float> rootsOf2(int a, int b, int c)
                                                  throws (noRealRoots)
{
        float disc ;
        if (disc < 0) throw noRealRoots;
        float routeDisc ;
        routeDisc = sqrt(b*b-4*a*c);
        float r1, r2;
        r1 = (-b+routeDisc)/(2*a);
        r2 = (-b-routeDisc)/(2*a);
        return std::make_pair(r1, r2);
}
```

code 4-1: The roots of a quadratic in C++

The scoping rules of C++ ensure that the variables disc, r1 and r2 are local to the function `rootsOf`. That the computational saving is marginal is not an issue; the technique is widely applicable and can be generalised and so effected by an interpreter, at least for function calls, using the technique known as memoisation.

The second benefit is of greater significance; the use of names for, in this case, expressions allows logical abstractions which provide strategies, such as procedural abstraction and functional decomposition, for the human designer to manage almost arbitrarily complex problems. This technique allows us to express naturally the logic of our solution.

The added expressiveness that derives from using local definitions is such that virtually all languages support their use; this ability also exists in the Z notation, not to support better execution, but to enhance the expressiveness of the notation. The mechanism for using local binding is obviously notation/language specific, but a brief examination of that mechanism in Z, in Haskell and in Lisp will establish the congruence of the three.

The simplest scenario that will allow a valid use has been chosen; each uses syntactically correct 'code', though in the latter two cases, it is probably not the best way to capture the logic.

The scenario is a library which models `reservations` as a function from a book descriptor to a sequence of borrowers; in each case the local 'variable' `reservers` is bound to the sequence of reservers *for this book.*

```
┌ReserveBook──────────────────────
│b? : BOOK
│m? : BORROWER
│reservations, reservations' : BOOK ⇸ seq BORROWER
│r! : success | alreadyReservedByThisBorrower
├──────────────────────
│( let reservers == reservations b?
│      •  ( m? ∈ ran reservers    ∧
│           reservations' = reservations ∧
│           r! = alreadyReservedByThisBorrower )
│                          ∨
│         ( m? ∉ ran reservers    ∧
│           reservations' = reservations ⊕ {b? ↦ (reservers⁀<m?>) } ∧
│           r! = success )
│ )
└──────────────────────────
```

Figure 4-1 : ReserveBook in Z

The use of a local binding in Haskell also involves a let clause, with a similar structure.

```
type Loans = [(Book, Borrower)]
type Library = (Book -> [Borrower],Loans)
type Report = String

reserveBook :: Library -> (Book , Borrower) -> (Library,
                                                  Report)

reserveBook  lib (b, m)
  = let
      (reservations, loans) = lib
      reservers = reservations b
    in
      if (m elem reservers)
      then
         (lib, "alreadyReservedByThisBorrower")
      else
         let
            newReservers = reservers ++ [m]
            newReservations =
                     reservations `override` [(b,newReservers)]
            newLib = (newReservations, loans)
         in
            (newLib, "success")
```

code 4-2 : reserveBook in Haskell

```
(defun reserveBook (lib b? m?)
  (let*
     ((reservations (first lib))
      (loans (second lib))
      (reservers (applyfn reservations b?)))
   (if (member m? reservers)
      (list lib "alreadyReservedByThisBorrower")
      (let
        ((newReservers (append reservers (list m?)))
        (newReserverations (override reservations `((,b? , newReservers))))
        (newLib (list newReservations loans)))
       (list newLib "success")))))
```

code 4-3 : reserveBook in Lisp

As was suggested, more natural versions could easily be written in both Haskell and Lisp, but the point here is to establish the close correspondence between all three, and more precisely, the existence in the programming languages of such similar constructs to that of the Z. In each case the let structure consists of the keyword `let`, a sequence of declarations of the objects being brought into local scope, followed by the expression to be evaluated in the augmented scope.

This close correspondence derives from the expression-based nature of the Z notation which is so fundamental to applicative languages, though surprisingly for a model-based specification language, Z is virtually free from the notation of updating state; the transformations concerned with decoration (as in object and object') explicitly reference two discrete objects, which need individual declarations and which just happen to have similar names and a convenient declaration shorthand.

It is not just this last characteristic that suggests a functional approach to the problem; the manipulation of expressions is by nature a recursive process and consequently, the implementation language chosen must support recursion. This requirement is straightforward to satisfy, but functional languages are explicitly designed, and consequently optimised, to handle recursion efficiently.

The exploratory programming approach can also be described as incremental development; even if the ultimate objective is known, the best route towards it may not be and in this area of ill-defined problems, it is important to maximise potential flexibility, to be best able to circumvent unforeseen problems, should they arise. A feature of the functional programming approach that contributes significantly to flexibility is the property of referential transparency. A consequence of referential transparency is that a function can be developed that will behave reliably, without reference to any objects or values other than its arguments; this allows the developer to provide incrementally more functionality, *that may or may not be useful.* A perception of the longer term objective will influence the choice of what behaviour should be developed, but not in the goal-driven, sequential way that is characteristic of and appropriate to better-defined problems. Notwithstanding the more usually valued benefits of referential transparency, namely the ability to formally reason with functional programs, it is also invaluable in supporting the incremental development of amorphous software systems.

There are so many instances of 'good-fit' matches that we can restrict the remainder of this review of correspondence to the consideration of the 'structural' match that by this point might be expected. The key abstraction mechanisms of functional programming, such as the treatment of functions as first-class data objects and their use in higher-order functions, are the same techniques from which Z derives much of its expressiveness; what [Spivey 1992] (Chapter 4) describes as *The Mathematic Tool-Kit,* with the functionals[7] such as relational composition, function inversion etc., is the same discrete mathematics that underpins functional programming.

$[X]$

$\bigcap : \mathbb{P}(\mathbb{P}\,X) \rightarrow \mathbb{P}\,X$

$\forall A : \mathbb{P}(\mathbb{P}\,X) \bullet$

$\quad \bigcap A = \{x : X \mid (\forall S : A \bullet x \in S)$

```
genIntersect :: Ord a => [[a]] -> [a]
genIntersect = foldl1 intersect
```

Figure 4-2 : generalised intersection in Z and Haskell

[7] This use is after [Backus 1978] and describes the connectives used to 'glue' functions together; they are also higher-order functions.

The Generalised intersection $\bigcap$ can be defined as in Figure 4-2 . An alternative description would be "the set of sets is reduced with the binary operator $\cap$ (set intersection)"; `reduce` is the list reduction operator in Lisp, or `fold` in Haskell. This converts directly to the executable code that models this behaviour, `genIntersect`.

Perhaps the clinching argument for adopting a functional perspective is more selfish.

> *"For programmers also, there is a tremendous gratification in frequently being able to express application theory or design ideas directly as program text, without the distracting details of an exactly ordered series of steps or accompanying sequence of memory operations"*
>
> [Runciman and Wakeling 1995] p216

4.2.2 The case for C++

A decision regarding choice of development paradigm is not the same as the choice of an implementation language; to illustrate this point it is worthwhile to examine what might be considered an unusual vehicle for a functional approach. C++ has evolved to support the development of software systems using an object-oriented approach, but that is not the only paradigm supported by what is now a mature and coherent language; particularly since the incorporation into the C++ standard [ISO/IEC 14882 1998] of what was originally, and is usually still referred to as, the Standard Template Library [Stepanov and Lee 1994]. Developments in the area of generic programming caused Stepanov and Lee to research and build an extensive library of generic containers and a set of algorithms that operate on those containers. These were adopted , largely unchanged, in the C++ Standard and for those developers that choose to use them, C++ now has the tools that allow a largely functional style to be adopted.

Given the recognition of the STL container class map as a function and the multimap as a relation, the Mathematical Tool-kit functions, such as dom, ran, relational inverse, relational image, can all be readily coded; Appendix A (STL/C++ structures) contains C++ code for these operations. However the usefulness of this approach is likely to be in the implementation of specifications, rather further along the development process than the stage of Requirements Capture and Validation with which we are predominately concerned. The unary functional composition operator o , though defined in the C++ standard as compose1 is not in fact implemented by Visual C++ version 6; its definition, in code 4-4, is illustrative of both the strengths and weaknesses of C++ as the implementation language of choice.

```
template <class Op1, class Op2>
unary_compose<Op1, Op2> compose1(const Op1 & f, const Op2 & g)
   {return (unary_compose<Op1, Op2>(f, g));};

template<class F_type, class G_type>
class unary_compose
    : public unary_function< F_type::argument_type, G_type::result_type>
{
  public:
    unary_compose(const F_type & f, const G_type & g) :f1(f), g1(g){};
    result_type operator()(const F_type::argument_type & y) const
           {return (g1 (f1(y)));};
  protected:
    F_type f1;
    G_type g1;
    };
```

code 4-4 : Unary function composition in C++

code 4-4 does indeed implement functional composition, but it is notably dense code that is non-trivial to implement; much of the obscurity derives from the need to use both inheritance and templates to support the polymorphism needed for the definition of higher-order functions. The weakness of this solution is highlighted by the marked contrast with the single line Lisp and Haskell equivalents, see code 4-5.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x   = g (f x)


(defun compose (f g)
       (lambda (x) (g (f x))))
```

code 4-5 : Function composition in Haskell and Lisp

4.2.3 The case for Haskell

Thus far, there has been no distinction made between any of the functional languages that might be chosen; both Haskell and Lisp have been invoked as preferable to C++, but they have been dealt with as if there was nothing to choose between them. This would certainly be the case for the implementation of the 'tool-kit' functionality, and Haskell has been shown to be a viable path to take in this regard [Sherrill & Carver 1993].

It is in regard to the larger context into which the toolset component must integrate that potential difficulties can be envisaged. At an early stage of the overall project a decision was taken that the toolset would be developed and, potentially deployed, using the Microsoft Windows operating system; this was primarily to maximise the user base that might develop should the toolset be genuinely useful. Few disadvantages were considered to derive from this decision, but one effect was to largely preclude Haskell from consideration; this was a tenable strategy as equally good, or better, alternatives to Haskell were available. A similar development using a totally functional subset of Lisp had successfully been undertaken [Hibberd 1990]; this experience demonstrated the feasibility of using Lisp, but it is worthwhile considering the characteristics that underpinned this confidence.

4.2.4 The case for Lisp

Though originally designed as a functional programming language [McCarthy 1960], Lisp has evolved many features that disqualify it from such a description; to name a few, these include the iterative and loop constructs (`loop`, `dotimes`, `progn` etc.), the destructive list manipulators (`nconc`, etc.), the side-effecting output functions (`princ`, etc.) and a plethora of state-setting possibilities centred around `setf`. Notwithstanding these, the basic data structure, the list, is the core structure of functional programs and the underlying style of programming remains prefix function application. It is quite straightforward to develop significantly large Lisp programs that use only the 'functional programming' features of Lisp – this subset of the language has been called FLisp [Glaser *et al*1984] and would be the basis of the expression-level functionality that implements Spivey's Mathematical Tool-kit. All of the necessary characteristics thus far identified can be found in FLisp; how this functionality might be 'bundled' and made available to a user does raise a number of new, though secondary, considerations. The implementation of an 'environment' or at least an interface is considered in Section 4.4; this a much better understood problem in many respects and its development using more conventional strategies would appear straightforward. That Lisp supports a variety of development paradigms as diverse as Object-Oriented and functional should ensure that an appropriate one is available when required. In fact CLOS, the Common Lisp Object System [Keene 1989], was available throughout the life of the project, but was deemed inappropriate. The close correspondence with Z would not be enhanced, more likely it would be undermined, by introducing a framework in ZAL that is not naturally part of Z.

The benefits of choosing Lisp as the development vehicle for, at least, the expression level functionality can be reprised as :

- Lisp is a functional language;
- Lisp supports incremental development;
- the Lisp chosen[8], though ultimately compiled, presents an interpreted interface to the user, with a command line and the source-level debugging support characteristic of such an integrated environment. These facilities make for a productive and usable development environment ;
- notwithstanding the above, the production code is compiled, with the attendant performance benefits of this ;
- Lisp can model non-deterministic choice [Graham 1994]p 297. This ability may or may not be required ;
- the access to the evaluator provided by Lisp provides an easy and accessible way to manipulate expressions, in whatever representation is appropriate. For example to model the scoping rules of the Z notation, an environment must be constructed and managed that respects the appropriate rules regarding the visibility of objects; what might be a free variable in an individual schema, may be brought into scope by the inclusion or composition of another schema. Given the interactive use of the animator envisaged in the Requirements Validation process, the ability to manipulate schema expressions at a source-code level is essential;
- Common Lisp is a standard(ised) language, so potential portability will be maximised ;
- the provision of a visual development environment and the library to support the development of 'visual' Lisp applications will facilitate the construction of a developer's interface
- experience suggests Lisp is the best possible vehicle for this endeavour, given that

[8] Allegro Common Lisp version 3.0.2

"Lisp is for building organisms - imposing, breathtaking, dynamic structures built by squads fitting fluctuating myriads of simpler organisms into place The discretionary exportable functionality entrusted to the individual Lisp programmer is more than an order of magnitude greater than that to be found within Pascal enterprises."

A J Perlis in Foreword to [Abelson et al 1985]

4.3 How the execution engine works

The division between the expression level functionality and the mechanism by which that functionality is used to animate any particular specification remains explicit in its implementation; the 'Mathematical Tool-kit' is converted from the discrete mathematics version presented in [Spivey 1992] into an equivalent Lisp source code version. The structural elements of the discrete maths, such as relations, sets, functions and sequences are modelled as Lisp lists with an explicit tag – a so-called *manifest* type [Abelson *et al* 1985].

If the animation is viewed as source code executing in some environment, the problem becomes to some extent one of the implementation of a "language" with particular execution semantics; the management of an execution context -principally a name space- is in fact simplified by the "update" semantics adopted. Given that updating is foreign to Z, the use of decorated names, such as *value'*, to indicate post-states makes explicit a potential update; this is in fact a little simplistic as there may be multiple references to a decorated name in a given scope and resolving which are potential updates is not necessarily straightforward.

4.3.1 ZAL data objects

The types of a Z specification are usually a combination of the basic types such as $\mathbb{Z}$, or more usually $\mathbb{N}$. and the given sets particular to that specification. The integer types are straightforward, as $\mathbb{Z}$ maps readily onto the FIXNUM, or BIGNUM, Lisp primitive types; the BIGNUM type provides arbitrary precision integers, so is in fact a better match then the integer types found in most programming languages.

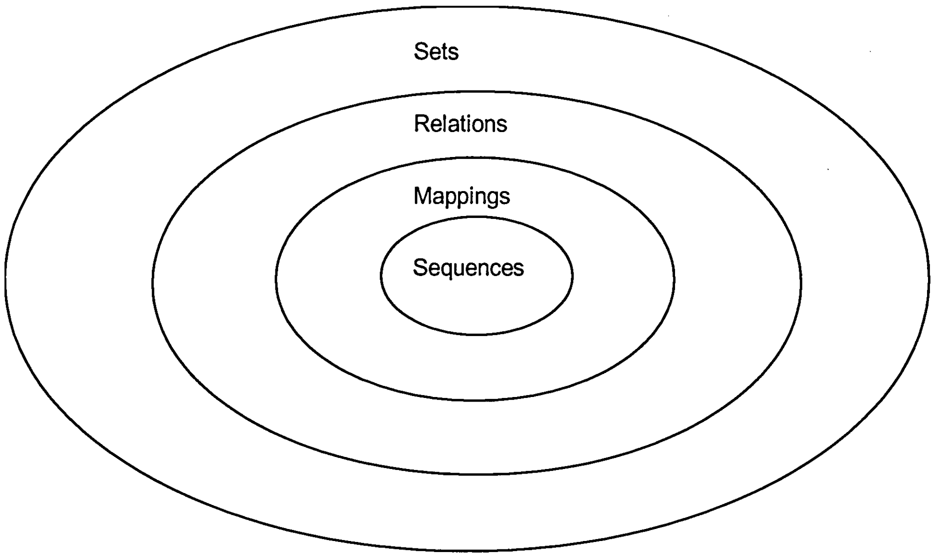


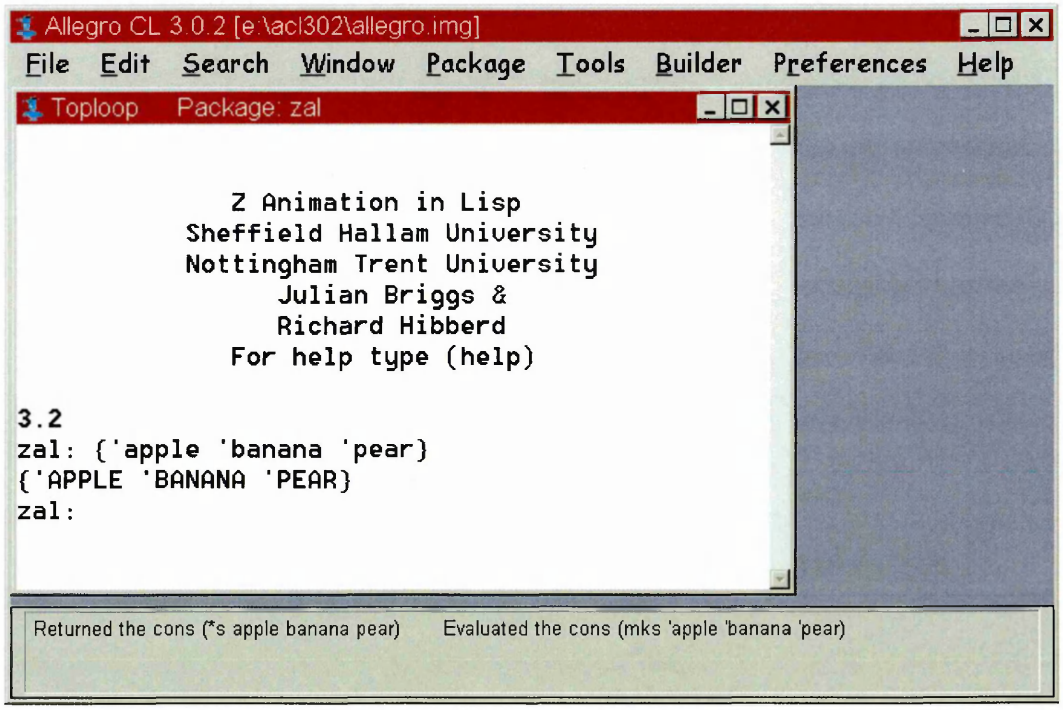
Figure 4-3 : The Z tower of types

These types will also form the components of more complex types that are formed by combining them into sets, relations etc. These latter structuring collections form a "tower of types" [Abelson *et al* 1985] (p151), which greatly simplifies their manipulation. In essence, the tower is a hierarchy where each type has at most one supertype and at most one subtype; the manipulation might involve coercion into different types from the tower. In this case the relationship is particularly straightforward, since all sequences are mappings, all mappings (including sequences) are relations, and all relations are sets; this gives a relationship more akin to subsets and supersets, as in Figure 4-3.

The consequence of this relationship is that type coercion, which can involve the loss of detail in the transformation, is not necessary; it is sufficient to determine the appropriate 'status' of an argument (i.e. "is it a relation?") at the point at which there exists such a requirement. This is generally implemented as a pre-condition, using predicates that examine the arguments.

The basic constructs of set, relation, mapping and sequence are each represented as Lisp lists of (potentially complex) types, with a tag field prepended to indicate the 'type'. This gives the following, apparently inelegant, representation of –say- the set {*apple, banana, pear* } as the list (*S APPLE BANANA PEAR); the *S is the tag field indicating a set and the capitalised 'strings' the Lisp print-names of the symbols APPLE BANANA and PEAR. This list is in fact the internal representation of the set and unsurprisingly it is possible to re-format this format into a more usual sophisticated form. That this re-formatting functionality can be inserted into the Top-Loop of the Lisp environment is more unusual; together with the use of developer-defined readtables, to accept the same 'sophisticated' syntax, it is possible to provide a more natural interaction, even at the command prompt, which is also redefined as in Figure 4-4

The read-eval-print behaviour of the command prompt was chosen as the initial interface to the execution engine; this gave initial accessibility and offered the potential to generalise the developed functionality to command strings that are generated dynamically. There are indications in the status bar of Figure 4-4 of the Toploop processing being performed - the entered string {′apple ′banana ′pear} is converted, using the opening brace character '{' as a macro-dispatch character, into the list, or cons, `(mks 'apple 'banana 'pear)`, which is the form actually evaluated by Lisp; this form returns `(*S APPLE BANANA PEAR)`, which is re-formatted and which then matches the original, except that is capitalised.


Allegro CL 3.0.2 [e:\acl302\allegro.img]
File Edit Search Window Package Tools Builder Preferences Help
Toploop Package: zal

```
                Z Animation in Lisp
             Sheffield Hallam University
             Nottingham Trent University
                   Julian Briggs &
                   Richard Hibberd
                For help type (help)

3.2
zal: {'apple 'banana 'pear}
{'APPLE 'BANANA 'PEAR}
zal:
```

```
Returned the cons (*s apple banana pear)     Evaluated the cons (mks 'apple 'banana 'pear)
```

Figure 4-4 : 'Natural' I/O

The actual symbols `apple` `banana` and `pear` must be "quoted", i.e. prefixed with the single quote character; this prevents their evaluation, which is the desired behaviour as they are simply symbols being used for their symbol-name. An alternative is to enter the values as strings, in quotation marks; these are not then symbols but simple string values, which evaluate to themselves. The input string in this case would be {"apple" "banana" "pear"}; this style is considered to be less natural and tends to raise in the mind of the user the unnecessary question of the string *representing* a physical or logical entity, rather than actually *being* that entity, and so was not adopted even though the chosen style does require the use of the quote character.

Table 2 has details of the input, output and internal representations of each of the type constructors

	Input format	Output format	Internal representation
Set	{'a 'b 'c}	{'A 'B 'C}	(*S A B C)
Relation	{#('p 'pear) #('p 'plum) }	{ #('P 'PEAR) #('P 'PLUM) }	(*R (P PEAR) (P PLUM))
Function (mapping)	{#('a 'apple) #('p 'plum) }	{ #('A 'APPLE) #('P 'PLUM) }	(*M (A APPLE) (P PLUM))
Sequence	<'apple 'pear 'plum>	<'APPLE 'PEAR 'PLUM>	(*Q (1 APPLE) (2 PEAR) (3 PLUM))

Table 2 Representation of collections

A more contrived example may demonstrate the purpose of the tag which determines the default format of display in the environment, which reflects the format used when the objects were constructed; this is important since the user would not expect the data objects to be implicitly transformed. For example a function from integer to string perhaps representing the mapping from a product code to its name might be instantiated `{#(1 "product 100") #(2 "product 200")}`; this could be tagged as a set, as a mapping, or a sequence. However each of these has distinctive (and in this case potentially misleading) output representations, as the tags are not normally displayed; this artificial example could legitimately also be displayed as a sequence, i.e. <"product 100" "product 200">, or as a set.

4.3.2 ZAL source as Extended Lisp

The development of the particular ZAL syntax has also been incremental; at any point, the 'language' has encompassed both the current subset of Z and also those structures and operators which are not yet fully implemented, but which can be constructed using a combination of existing elements and primitive Lisp code, as opposed to the extended Lisp that characterises ZAL. This primitive, raw Lisp code has been used to model the appropriate functionality and later, once the requisite behaviour has been established, as the basis for the actual implementation of the feature. This approach has facilitated the smooth evolution of the functionality.

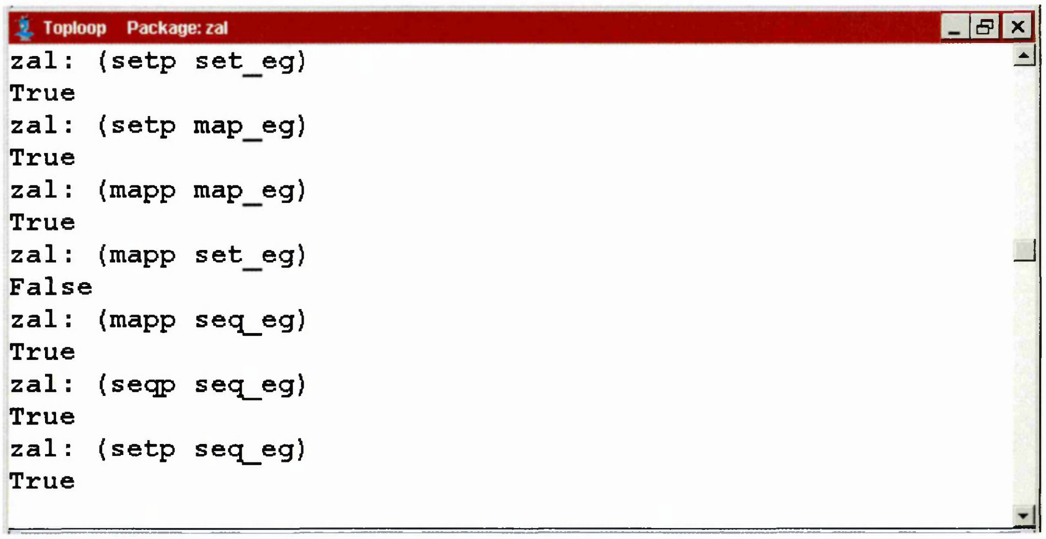
Another, less important, consideration has been the efficiency of the execution; for example, the implementation of the range restriction operator $\rhd$ began as

$$R \rhd S \Leftrightarrow (S \lhd R^{\sim})^{\sim}$$

This implementation yields *$O(n^2)$* complexity, which was generally considered sufficiently poor to warrant a more sophisticated implementation, if one was available, which in this case involved a single pass over the relation, with corresponding *$O(n)$* complexity, but some less clarity. In contrast, relational image $(\!|\,|\!)$ remains implemented as $R(\!|S|\!) \Leftrightarrow \mathit{ran}\ (S \lhd R)$, since this is still *$O(n)$*.

4.3.3 An error reporting regime

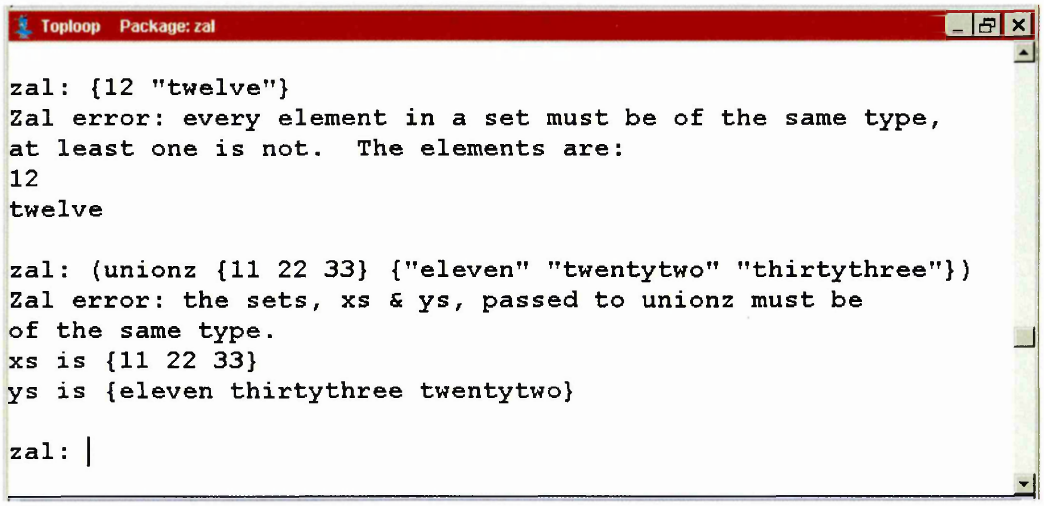
The data validation of the arguments to almost all the ZAL functions is performed "locally", that is as part of the ZAL function; this can sometimes lead to duplicated computation but supports a more natural feedback style as the testing is performed when the user "naturally" expects, at run-time and in the execution sequence. There are ZAL predicates to test objects as sets, relations, functions and sequences and these are respectively `setp, relp, mapp, seqp`; their use can be seen in Figure 4-5.



```
zal: (setp set_eg)
True
zal: (setp map_eg)
True
zal: (mapp map_eg)
True
zal: (mapp set_eg)
False
zal: (mapp seq_eg)
True
zal: (seqp seq_eg)
True
zal: (setp seq_eg)
True
```

Figure 4-5 : ZAL predicates

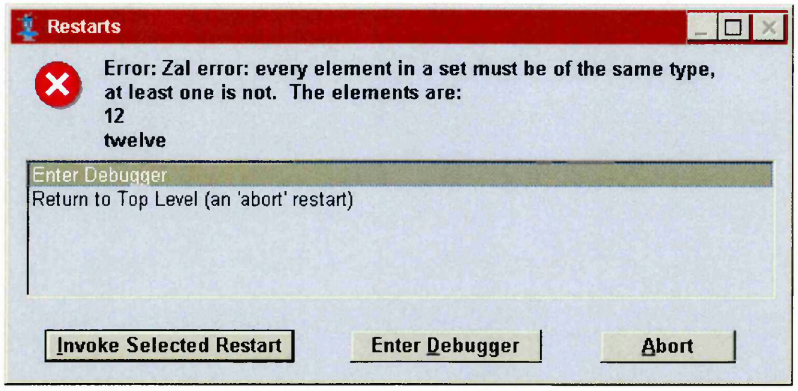
This verification of arguments can be seen wherever the possibility exists of the user presenting inappropriate data values; attempting to build a set of differently typed elements will fail, as will the union of two differently typed sets.



```

zal: {12 "twelve"}
Zal error: every element in a set must be of the same type,
at least one is not.  The elements are:
12
twelve

zal: (unionz {11 22 33} {"eleven" "twentytwo" "thirtythree"})
Zal error: the sets, xs & ys, passed to unionz must be
of the same type.
xs is {11 22 33}
ys is {eleven thirtythree twentytwo}

zal: |
```

Figure 4-6 : Argument verification and error reporting

In fact in normal use, these reports are directed to an error-reporting window (Figure 4-7 : Error reporting dialogue); the top loop, while available, is the province of experienced users.


Restarts

Error: Zal error: every element in a set must be of the same type, at least one is not. The elements are:
12
twelve

```
Enter Debugger
Return to Top Level (an 'abort' restart)
```

Figure 4-7 : Error reporting dialogue

It might appear that the error handling implemented within ZAL is unsophisticated in that no use is made of exception-passing to facilitate recovery from errors; the error is reported and the user offered the choice of an 'abort' restart or use of the Lisp Debugger. In fact the Debugger offers the full functionality of the underlying environment, together with inspection and edit facilities for the context of the error and recovery from it. This is actually rather more functionality that the intended user of the system would normally need; choosing an 'abort' restart will cause the stack unwinding needed, together with the de-allocation of local objects, which is the major benefit of exception handling regimes, and this would be the choice expected in almost all situations. This rather binary view of the execution is consistent, in style at least, with the execution of Z as interpreted here – an execution is successful or not.

4.3.4 Expression level manipulations

Having established the basic data components of the ZAL subsystem, the operations that manipulate those data components can now be examined to complete the description of the underlying functionality. These operations will very much correspond to the discrete mathematics operators that are used to capture behaviour at an expression level within Z schema.

The toolkit elements and their ZAL equivalents can be seen in Table 3.

The following Z constructs have been implemented.

Z construct	**Description**	**`ZAL function`**
#	set cardinality	`card`
(,)	pair construction	`#(   )`
$<$	relational operator	`#\<` [9]
$<>$	sequence constructor	`< >`
$=$	equality testing / binding operator	`Eqz`
$\mathbb{P}$	powerset	`powerset`
$\mapsto$	maplet	uses 2-tuple
$\in$	set membership	`is-mem`
$\notin$	set non-membership	`is-not-mem`
$\subset$	proper subset	`psubset`
$\subseteq$	subset	`subset`
$\cap$	set intersection	`inter`
$\cup$	set union	`unionz`
$\setminus$	set difference	`setsub`
$\bigcap$	distributed (generalised) intersection	`inter-dis` or `gen-inter`
$\bigcup$	distributed union	`union-dis` or `gen-union`
⨾	relational composition	`Rel-compose`
$\lhd$	domain restriction	`domres`
⩤	domain subtraction	`domsub`
$\rhd$	range restriction	`ranres`
⩥	range subtraction	`ransub`
$\oplus$	function override	`overrride`
$\frown$	sequence concatenation	`concat`
$\exists$	existential quantification	`exist`
$\forall$	universal quantification	`forall`
⦇⦈	relational image	`Rel-image`
$\exists_1$	unique existential quantification	`exist-one`
$\vee$	logical disjunction	`Or`

[9] The character '<' is redefined to facilitate 'natural' sequence input, so the relational operator must be an escape sequence; this is not an issue, since this is generated by TranZit.

$\wedge$	logical conjunction	`And`
$\Rightarrow$	logical implication	`imply`
disjoint	disjoint	`disjoint`
distributed disjoint	distributed disjoint	`disjoint-dis`
dom	domain	`dom`
	function application	`applyz`
head	head	`head`
ran	range	`ran`
$\sim$	relational inverse	`inverse`
<>	sequence constructor	`< >`
{}	set constructor	`{ }`
tail	tail	`tail`

Table 3 : Mathematical toolkit equivalents in ZAL

4.3.4.1 General expressions

The style of modelling that has been adopted, with a close correspondence between the Z Mathematical Toolkit and the ZAL function that implements it, gives rise to a bottom-up development style; given the representation of data components described in Section 4.3.1, the coding of this functionality, with the exception of the quantifications and the comprehensions, is for the most part straightforward.

4.3.4.2 Testing for equality and binding values

There are two ways in which the = symbol is used in Z specifications. It can be either the equality testing operator as in the constraining predicate *state = state'* ; this would indicate that the object *state* must not be changed (the decoration ' is used to indicate an after-state). Alternatively the symbol can be used to generate new 'bindings' for objects as with

$$\mathsf{someFn}' = \mathsf{someFn} \oplus \{\mathsf{d} \mapsto \mathsf{r}\}$$

which generates a value for the after-state of the object *someFn* using the function override operator. ; this can be considered a binding within the scope of the schema expression. This twin rôle presents no difficulty in the Z as the second use can still be seen as that of equality testing, but in the declarative sense of constraining the after-state values to satisfy this predicate while not making deterministic any requirements regarding sequencing or indeed particular values for the components; however it constitutes an issue of some significance in this undertaking.

4.3.4.2.1 Execution sequence

In so much as our execution will constitute a single, possibly deterministic, representation of a possibly non-deterministic specification, and that the outcome of that execution, if terminating, is ultimately a truth value, that we adopt by default a sequential, down-the-page regime is simply a choice that may or may not be effective when applied to a particular specification. Empirically, it has been found to be a sound approach in many cases, since the writers of Z specifications tend to adopt a pre-condition, update, post-condition sequencing in their specifications. Nonetheless, it would be un-necessarily constraining to require this explicit sequencing, and so strategies have been developed to minimise the set of valid specifications which cannot be animated.

To this end, at execution-time the context of the equality expression is examined to determine the precise use being made of the operator; this was initially performed when the schema object was defined, but this approach cannot manipulate predicates from included schema, which require that this examination be deferred until the actual, expanded schema expression is available for analysis. This analysis of the predicates of this expanded object reveals whether the operator is being used un-ambiguously (either to bind a value or to test for equality) or if the use is ambiguous; an ambiguous use might derive from a schema with a decorated object being used in more than one predicate or expression.

This analysis is used to broadly categorise the predicates and to sequence them on this basis. The expectation was that this would be too naïve a solution and that more sophisticated strategies would be needed to resolve this issue; this has in fact not been the case and this approach has not been constraining, though further research may yet be appropriate.

4.3.4.3 The question of quantifiers

This section will examine the mechanism by which existential and universal quantifications are modelled in ZAL; implicit set generation, or set comprehensions, will emerge as a similar, though not congruent domain, with an equivalent set of issues and solutions.

As was described in Section 3.2.1, the possibility of a potentially infinite search space has been resolved *before* ZAL is required to evaluate expressions defined on it; this is achieved by identifying a constraint from the Z expression, or if that fails, by user intervention.

A simple example will highlight the mechanism employed; it features the existential quantifier but the detail is identical for a universal quantification. The predicate

$$\exists j : \mathbb{N} \mid j \in 0..10 \bullet j * j = 81$$

gives rise to the ZAL code, generated automatically

```
    (exist j (mks 0 1 2 3 4 5 6 7 8 9 10) (= (* j j) 81)))
```

`exist` is implemented as a macro which is expanded into a further macro call to `x-st` before the arguments are evaluated; this two stage expansion facilitates recursion for multiple quantified variables and can be seen in Figure 4-8

The original call

```
 (exist j (mks 0 1 2 3 4 5 6 7 8 9 10) (= (* j j) 81))
```

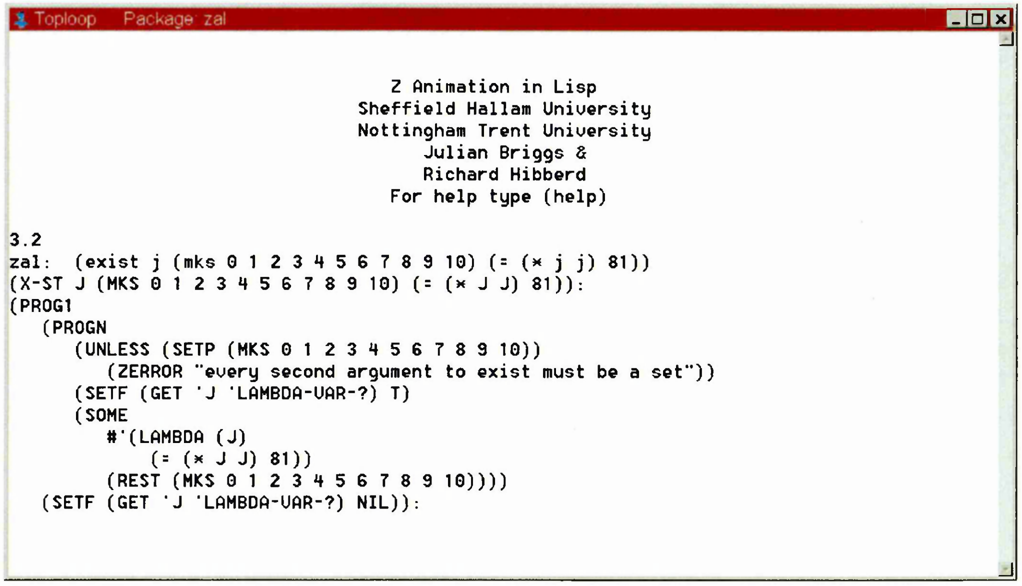
expands to

```
   (X-ST J (MKS 0 1 2 3 4 5 6 7 8 9 10) (= (* J J) 81))
```

This expression is again expanded to

```
(PROG1
  (PROGN
    (UNLESS (SETP (MKS 0 1 2 3 4 5 6 7 8 9 10))
      (ZERROR "every second argument to exist must be a set"))
    (SETF (GET 'J 'LAMBDA-VAR-?) T)
    (SOME
      #'(LAMBDA (J)
          (= (* J J) 81))
      (REST (MKS 0 1 2 3 4 5 6 7 8 9 10))))
   (SETF (GET 'J 'LAMBDA-VAR-?) NIL))
```

and this is the expression that is eventually evaluated.



```
                         Z Animation in Lisp
                      Sheffield Hallam University
                      Nottingham Trent University
                           Julian Briggs &
                           Richard Hibberd
                         For help type (help)
3.2
zal:  (exist j (mks 0 1 2 3 4 5 6 7 8 9 10) (= (* j j) 81))
(X-ST J (MKS 0 1 2 3 4 5 6 7 8 9 10) (= (* J J) 81)):
(PROG1
   (PROGN
      (UNLESS (SETP (MKS 0 1 2 3 4 5 6 7 8 9 10))
         (ZERROR "every second argument to exist must be a set"))
      (SETF (GET 'J 'LAMBDA-VAR-?) T)
      (SOME
         #'(LAMBDA (J)
             (= (* J J) 81))
         (REST (MKS 0 1 2 3 4 5 6 7 8 9 10))))
   (SETF (GET 'J 'LAMBDA-VAR-?) NIL)):
```

Figure 4-8 : Macro-expansion of an existential quantification

The processing is accomplished using the Lisp higher-order function `some` which determines if the function argument returns true when applied to any element of the list argument; also visible in the expanded Lisp/ZAL is error-checking code and code to identify and manage the name-space at execution. In fact when evaluated within the animation environment, rather than at the command line as here, the quantified variables, j in this case, are replaced by `gensyms` (automatically generated unique symbol names) after expansion but before execution.

The recursive macro-expansion performs a re-write equivalent to

$\exists$ i:I j:J $\bullet$ Q $\Leftrightarrow$ $\exists$ i:I $\bullet$ ($\exists$ j:J $\bullet$ Q)

Notice that the quantified expression is not the most general, lacking the usual constraint P in

$\exists$D | P $\bullet$ Q

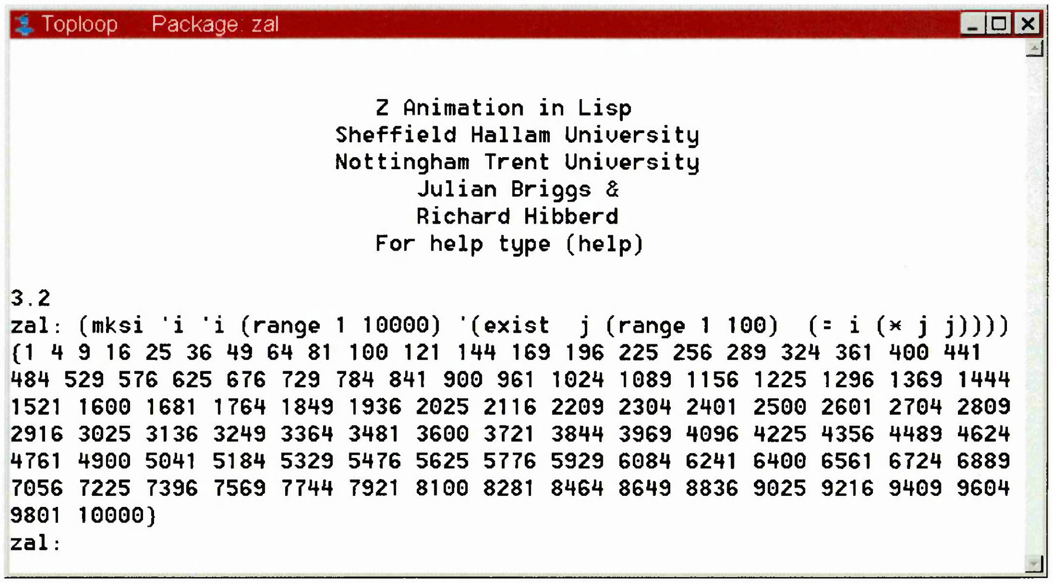
It is the case that constraining predicates P *either* constrain the search space, in which case they will have been 'absorbed' into the declaration D *or* they will have been rewritten using the equivalence

$$\exists D \mid P \bullet Q \Leftrightarrow \exists D \bullet P \wedge Q$$

There are a number of similarities between the quantifications described above and the generation of implicit sets, or set comprehensions; for both it is necessary to generate a set of candidate data and to test that data with a predicate. For a quantification, the result of the test generates the result of the quantification; for a set comprehension, the test is used to filter the candidate data, with the successful values being used to construct the set elements. An often-cited example of the *drawbacks* of executing specifications is the inability to handle this definition of the set of perfect squares

$$\{i : \mathbb{N} \mid \exists j : \mathbb{N} \bullet j^2 = i\}$$

It is absolutely the case that this will not directly transform into an executable form, since both the set and the natural numbers are infinite. However by constraining $\leq$ to some finite range, it will execute without further change; in particular, there is no need to introduce any algorithmic detail, as in Figure 4-9. Larger ranges can be searched, but these require the use of the non-pure `nrange` function to generate the candidate data.


Toploop Package: zal

```
                    Z Animation in Lisp
                Sheffield Hallam University
               Nottingham Trent University
                     Julian Briggs &
                     Richard Hibberd
                   For help type (help)

3.2
zal: (mksi 'i 'i (range 1 10000) '(exist  j (range 1 100)  (= i (* j j))))
{1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361 400 441
484 529 576 625 676 729 784 841 900 961 1024 1089 1156 1225 1296 1369 1444
1521 1600 1681 1764 1849 1936 2025 2116 2209 2304 2401 2500 2601 2704 2809
2916 3025 3136 3249 3364 3481 3600 3721 3844 3969 4096 4225 4356 4489 4624
4761 4900 5041 5184 5329 5476 5625 5776 5929 6084 6241 6400 6561 6724 6889
7056 7225 7396 7569 7744 7921 8100 8281 8464 8649 8836 9025 9216 9409 9604
9801 10000}
zal:
```

Figure 4-9 : Perfect squares defined with a set comprehension

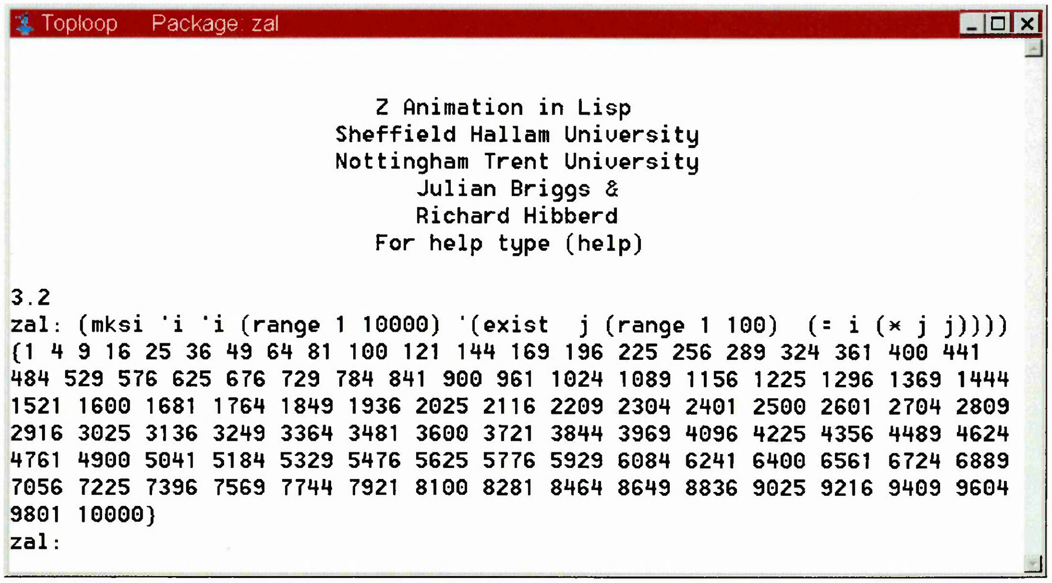
4.3.5 Schema definition

The initial approach to the creation of the schema object was to encapsulate the declarations and behaviour into a Lisp functional object – both a closure and a lambda expression were investigated. This generates small closures that can be readily manipulated and provides a mechanism for schema inclusion using nested calls to the execution function. This was also useful to support schema renaming and an explicit call to one schema from within another – this technique is used by Morgan in a classic specification of a telephone network in [Hayes 1993] as in the fragment

$\neg(\exists\ cons^0 : \mathbb{P}\ CON \bullet cons \subset cons^0 \wedge TN[cons^0 / cons]$

where TN is a schema object, as is the schema-renamed expression. Though effective in this particular situation, the technique limits the optimisations that can be made to those valid at an individual schema level, at the time of schema definition. More seriously it is not possible to establish which elements of a schema are visible at that point; the alteration of a schema that is already *included* by others will invalidate all of the *including* schemas.

This suggests that the correct point at which to encapsulate the declarations and predicates is at execution. This does entail a significantly larger computational object,



```
                    Z Animation in Lisp
                 Sheffield Hallam University
                 Nottingham Trent University
                      Julian Briggs &
                      Richard Hibberd
                    For help type (help)

3.2
zal: (mksi 'i 'i (range 1 10000) '(exist  j (range 1 100)  (= i (* j j))))
{1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361 400 441
484 529 576 625 676 729 784 841 900 961 1024 1089 1156 1225 1296 1369 1444
1521 1600 1681 1764 1849 1936 2025 2116 2209 2304 2401 2500 2601 2704 2809
2916 3025 3136 3249 3364 3481 3600 3721 3844 3969 4096 4225 4356 4489 4624
4761 4900 5041 5184 5329 5476 5625 5776 5929 6084 6241 6400 6561 6724 6889
7056 7225 7396 7569 7744 7921 8100 8281 8464 8649 8836 9025 9216 9409 9604
9801 10000}
zal:
```

Figure 4-9 : Perfect squares defined with a set comprehension

4.3.5 Schema definition

The initial approach to the creation of the schema object was to encapsulate the declarations and behaviour into a Lisp functional object – both a closure and a lambda expression were investigated. This generates small closures that can be readily manipulated and provides a mechanism for schema inclusion using nested calls to the execution function. This was also useful to support schema renaming and an explicit call to one schema from within another – this technique is used by Morgan in a classic specification of a telephone network in [Hayes 1993] as in the fragment

$\neg(\exists\ cons^0 : \mathbb{P}\ CON \bullet cons \subset cons^0 \wedge TN[cons^0 / cons]$

where TN is a schema object, as is the schema-renamed expression. Though effective in this particular situation, the technique limits the optimisations that can be made to those valid at an individual schema level, at the time of schema definition. More seriously it is not possible to establish which elements of a schema are visible at that point; the alteration of a schema that is already *included* by others will invalidate all of the *including* schemas.

This suggests that the correct point at which to encapsulate the declarations and predicates is at execution. This does entail a significantly larger computational object, since it must be built in totality rather than referencing previously defined objects, but this trade-off is well justified by the clarity and correctness of the model; furthermore it is consistent with the notion of schema inclusion as a textual expansion.

The schema definition process is consequently largely reduced to a call to a structure constructor (a structure is the Lisp record type), with initialising values for the various fields. As a convention, static data is held in the fields of the structure and the run-time flags and parameters are managed as property lists, which provide more flexible access facilities.

4.3.6 Schema level manipulations

Though it is technically feasible to adopt a purely functional approach to the manipulation of schema entities, rather than just the clauses of them, Z remains a state-based modelling notation; indeed the role of the *state* schema is to introduce those objects that constitute the state, together with the state invariants, the constraints that the state will always satisfy.

The putative functional approach to schema level manipulations would necessitate the passing of the state data into and out of an execution; such a system would be unusable without some mechanism to automate the generation and management of this data. Additionally, the *raison d'etre* of the whole project is to make accessible to a sponsor the details of a specification whilst minimising the technical skills required; it would be illogical to then require an understanding of perhaps monadic I/O and the functional paradigm. To this end, it is important that externally at least, the animation appears to maintain an ongoing state. The virtual proscription of global variables does not apply to the situation here; the state schema introduces the state objects of the specification and that state is *naturally* global. Furthermore, the global objects in question are in effect named values, and are not updateable in the conventional sense; pre- and post-states can been interpreted as separate values and the only 'updating' of the state objects is the promotion of some or all of the post-state values of some execution to be the current values of the state. *Between* executions, the values may be edited, but that can be interpreted as changing the 'input' data for the next execution; there exist command line facilities to update state data, but these are not part of the ZAL code generated automatically by TranZit.

The user's perspective of executing a schema will be examined in Section **0**, but a technical overview is in order to better provide a context for the current discussion. After a schema (name) is selected for execution, the process detailed below occurs:

- the executable object is constructed dynamically – this involves expanding recursively any referenced schemas and building these into a single object. These references may be by inclusion, conjunction, disjunction or negation;
- an environment is constructed for the execution, to include any objects that are in scope in the lifetime of the execution;
- any inputs required by the schema are collected, by interaction with the user;
- the executable object is evaluated in the constructed environment;
- the result of the evaluation, a truth value, is reported to the user. If the evaluation succeeded, any outputs are reported to the user, together with any other objects selected for display and the opportunity to promote the values of post-state objects is offered.

After a brief discussion of the role of the state schema, this process is examined in detail.

4.3.6.1 The role of schemas in Z and ZAL

The state schema is conventionally used to introduce the data objects of concern within a specification, and to constrain the values that those objects may take; this can be thought of as analogous to a variable declaration in a conventional programming language, which brings the declared object *into scope.* The predicate clauses of the state schema constrain the values, effectively defining what is or is not a valid state, and for this reason, the predicate of the state schema is sometimes known as the invariant. A novel aspect of the work is the practice of executing the state schema, or at least the invariant, which has the effect of verifying the current state. As will be examined in Section 5, this can be done in the normal way using the usual interface as in Figure 5-2 (page 106) and Figure 5-11 (page 125); it can also be automated so that a check is made during the promotion of post-state values, Figure 5-17(page 111).

Another use of a schema is as an abstraction, whereby the schema simply serves as a convenient name for a collection of data objects or predicates or both; this style of use can be found in Section 5.2.3 and Figure 5-13 (page 126), where the schema `Monitorvar?` serves simply to bundle a commonly used set of inputs. The schema is then included wherever that combination of inputs is required; this use is allowable because of the namespace rules that Z employs. An object can be introduced by a declaration in a schema or in a schema that is included in the original schema or in fact in more than one of these – the only requirement is that all the declarations must be the same, i.e. the object must be declared to be of the same type. In this respect Z differs from virtually all programming languages, which generally utilise a hierarchical scheme, with all occurrences being separate and at most one being visible at any point, and the remainder hidden or shadowed. As a consequence, this namespace must be explicitly managed, as is discussed in Section 4.3.6.7

4.3.6.2 Constructing the executable object

This phase is concerned with the creation of a Lisp lambda-expression, which is an un-named function expression that can be applied in a way identical to any other function application. The form of the expression is

```
#'(lambda args body)
```

where `args` is a possibly empty list of formal parameters to the expression and `body` is a symbolic expression, or s-exp, which calculates the return value of the function.

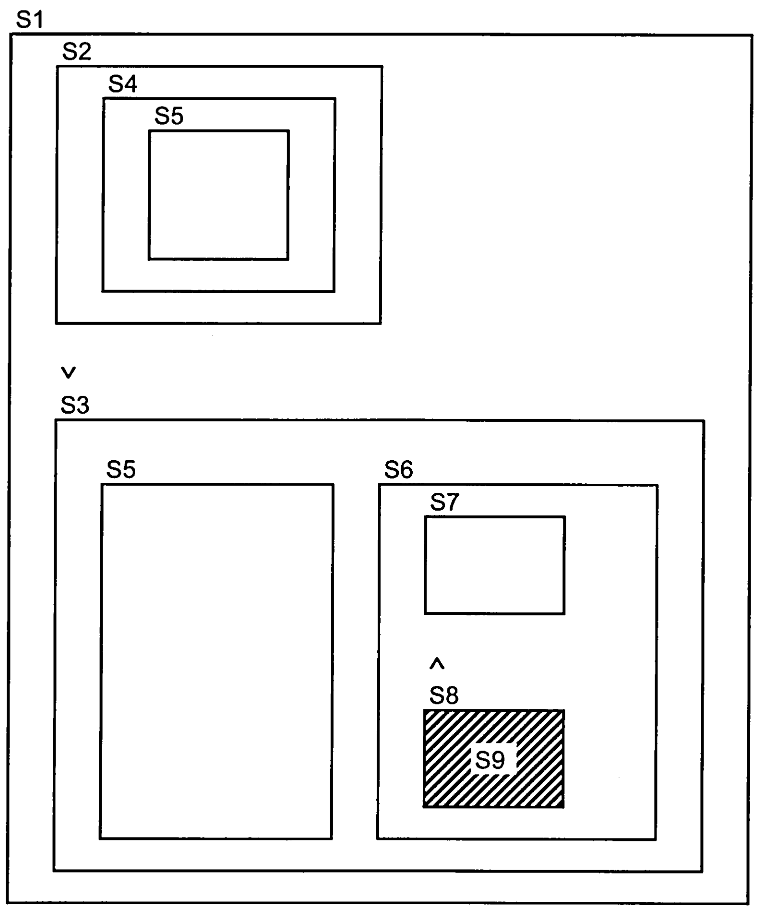
It is necessary to construct both `args` and `body` from the aggregation of all the schemas that are referenced directly or indirectly by the "executing" schema; to this end, the relationships are examined and their topology captured.

The references to other schemas all derive from the schema calculus; the particular set of operations supported are :

- schema negation
- schema inclusion
- schema conjunction and
- schema disjunction.

These schema operators determine the relationship between the elements that make up the executable object; the elements are the argument schema. suggests an organisation that will illustrate the mechanism by which both the declarations and the predicate of the executable object are constructed.

Given the relationships of and that schema S*n* has declarations D*n* and predicate P*n*, $D^{+}S1$,the expanded declarations of S1, can be expressed in terms of D1 and the expanded declarations of S2 and S3, and correspondingly, the expanded predicate of S1, $P^{+}S1$ in terms of P1 and the expanded predicates of S2 and S3.



$S1 \hat{=} S2 \lor S3$

S2 includes S4

S4 includes S5

S3 includes S5

S3 includes S6

$S6 \hat{=} S7 \land S8$

S8 negates S9

Figure 4-10 : An illustrative schema topology

More formally

$$D^{+}n \mathrel{\widehat{=}} Dn \cup$$

$$\left(\bigcup \left(\alpha\ D^{+}\,\mathrm{includedBy}(n)\right)\right) \cup$$

$$\left(\bigcup \left(\alpha\ D^{+}\,\mathrm{conjoinedBy}(n)\right)\right) \cup$$

$$\left(\bigcup \left(\alpha\ D^{+}\,\mathrm{disjoinedBy}(n)\right)\right) \cup$$

$$\left(\bigcup \left(\alpha\ D^{+}\,\mathrm{negatedBy}(n)\right)\right)$$

or

$$D^{+}n \mathrel{\widehat{=}} Dn \cup \left(\alpha\ D^{+} \left(\bigcup \{\ \mathrm{includedBy}(n)\ \ \mathrm{conjoinedBy}(n)\ \ \mathrm{disjoinedBy}(n)\ \ \mathrm{negatedBy}(n)\ \}\right)\right)$$

where   is based on [Backus 1978] and is the apply-to-all-set-elements operator defined

$$\begin{array}{|l}
[Y,Z] \\
\quad : (Y \rightarrow Z) \times \mathbb{P}\,Y \rightarrow \mathbb{P}\,Z \\
\hline
\forall\ f : (Y \rightarrow Z)\ ;\ S : \mathbb{P}\,Y\ \bullet \\
\qquad f\ S = \{\ y : Y \mid y \in S \bullet f\ y\ \} \\
\hline
\end{array}$$

This is a somewhat simplistic interpretation of the expansion of the declarations, which is consistent largely because the visibility of the objects has been validated by the TranZit component and so using the expanded declaration does not introduce elements that are accessed by, say, one element of a schema disjunction when they are declared in another.

The expression giving the expanded predicate is more complex, but can be expressed:

$$
\begin{aligned}
P^{+}(n) \triangleq \quad & Pn \wedge \\
& (\mathit{dist}\wedge \ (\alpha\ P^{+}\,\mathrm{includedBy}(n)\,)\,) \wedge \\
& (\mathit{dist}\wedge \ (\alpha\ P^{+}\,\mathrm{conjoinedBy}(n)\,)\,) \wedge \\
& (\mathit{dist}\vee \ (\alpha\ P^{+}\,\mathrm{disjoinedBy}(n)\,)\,) \wedge \\
& (\neg\ P^{+}\,(\mathrm{negatedBy}(n)\,)\,)
\end{aligned}
$$

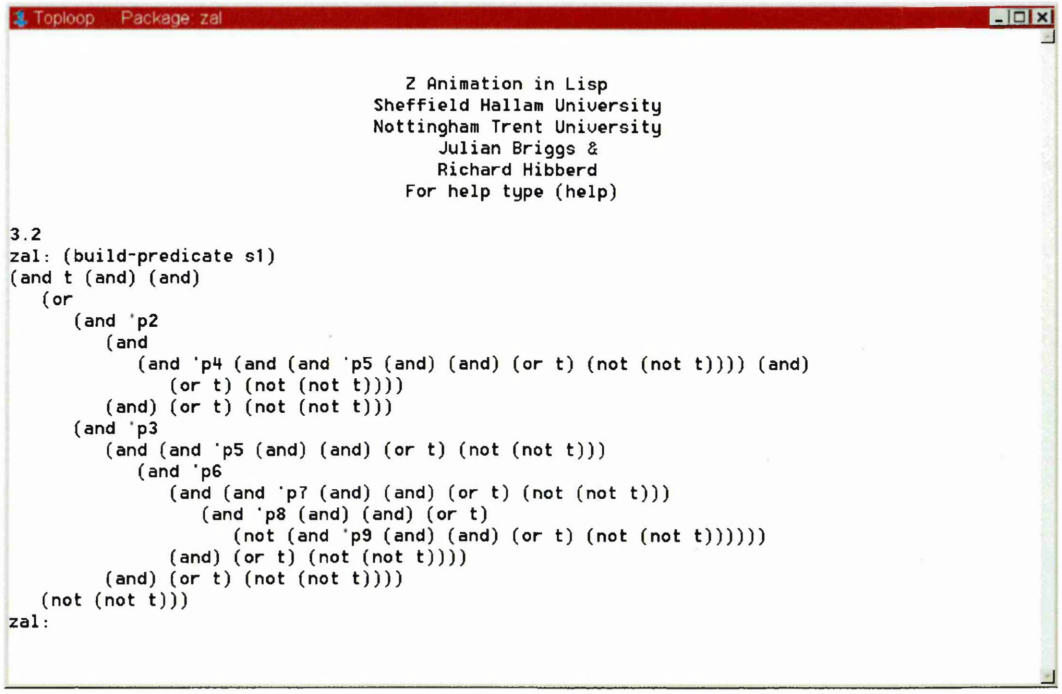
where $\mathit{dist}\wedge$, $\mathit{dist}\vee$ are distributed conjunction and distributed disjunction

respectively and are defined

$$
\begin{array}{|l}
\mathrm{dist}\wedge\ ,\ \mathrm{dist}\vee\ :\ \mathbb{P}\ (\mathbb{P}\ \mathrm{Bool})\ \rightarrow\ \mathbb{P}\ \mathrm{Bool} \\
\hline
\forall\ A\ :\ \mathbb{P}\ (\mathbb{P}\ \mathrm{Bool})\ \bullet \\
\qquad \mathrm{dist}\wedge\ =\ (\forall p\ :\ A\ \bullet\ p\ \ )\ \wedge \\
\qquad \mathrm{dist}\vee\ =\ (\exists p\ :\ A\ \bullet\ p\ \ )
\end{array}
$$

Figure 4-11 : distributed conjunction and distributed disjunction in Z

The expanded predicate of the topology of can be extracted from the executable object, see Figure 4-12



```
                              Z Animation in Lisp
                           Sheffield Hallam University
                          Nottingham Trent University
                              Julian Briggs &
                              Richard Hibberd
                              For help type (help)

3.2
zal: (build-predicate s1)
(and t (and) (and)
   (or
      (and 'p2
         (and
            (and 'p4 (and (and 'p5 (and) (and) (or t) (not (not t)))) (and)
               (or t) (not (not t))))
         (and) (or t) (not (not t)))
      (and 'p3
         (and (and 'p5 (and) (and) (or t) (not (not t)))
            (and 'p6
               (and (and 'p7 (and) (and) (or t) (not (not t)))
                  (and 'p8 (and) (and) (or t)
                     (not (and 'p9 (and) (and) (or t) (not (not t))))))
               (and) (or t) (not (not t))))
         (and) (or t) (not (not t))))
   (not (not t)))
zal:
```

Figure 4-12 : The expanded predicate of schema S1

A reduced topology – that rooted at S6 – generates the predicate below, which has been annotated to show the origin of each clause.

```
(and 'p6                                  predicate of S6
  (and                                    included by S6 – S7
    (and 'p7                              predicate of S7
       (and)                              included by S7 - nil
       (and)                              conjoined by S7- nil
       (or t)                             disjoined by S7- nil
       (not (not t)))                     negated by S7- nil
    (and 'p8                              predicate of S8
       (and)                              included by S8- nil
       (and)                              conjoined by S8- nil
       (or t)                             disjoined by S8- nil
       (not                               negated by S8
          (and 'p9                           – the predicate of S9
             (and)                        included by S9- nil
             (and)                        conjoined by S9- nil
             (or t)                       disjoined by S9- nil
             (not (not t))))))            negated by S9- nil
  (and)                                   conjoined by S6- nil
  (or t)                                  disjoined by S6 nil
  (not (not t)))                          negated by S6- nil
```

There exists some redundancy in this predicate which could be readily simplified, if necessary.

4.3.6.3 Constructing the environment

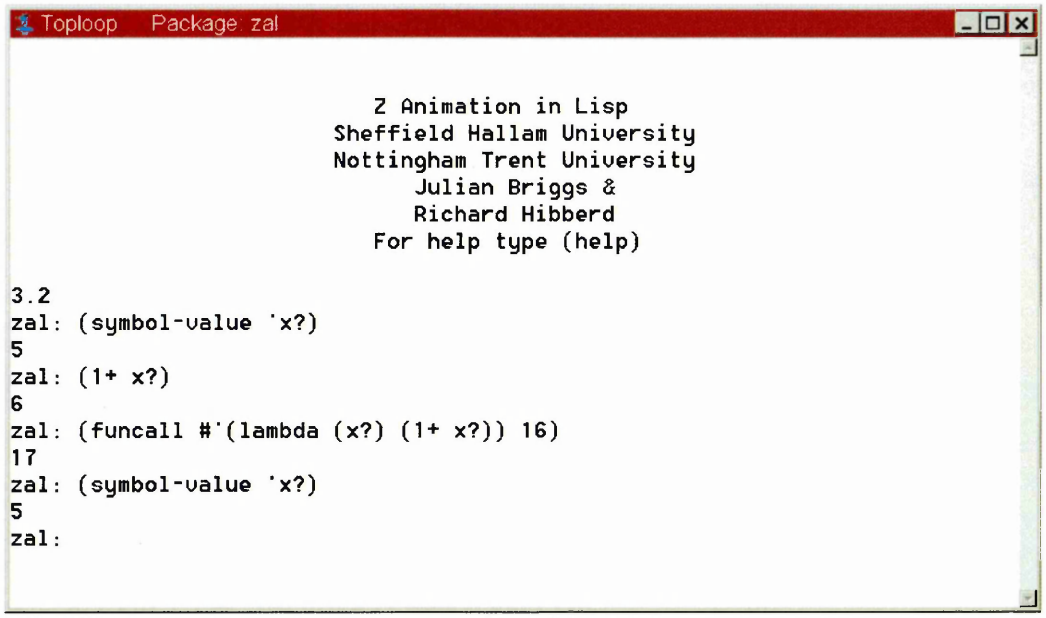
Each of the data objects that is in scope in the lifetime of the execution belongs to exactly one of the following categories:

- an input object
- an output object
- a locally declared object
- a state object that is declared in the state schema

Each of these is handled individually and the particular characteristics of each are discussed below; however it should be noted that only the state objects exist outside the scope of a schema execution.

4.3.6.3.1 Input objects

These are objects that are used to construct the actual argument list to the executable object; they are not a part of the execution itself since any references to the name within the body of the executable object refer to the actual parameter, as is suggested by Figure 4-13. Even if the input object also names an existing symbol, the value entered by the user is not the symbol's value, but is stored in a property list as specific to this execution. The arguments are all presented as values rather than symbols, so there is no possibility of updating these objects.



```
                    Z Animation in Lisp
                 Sheffield Hallam University
                 Nottingham Trent University
                      Julian Briggs &
                      Richard Hibberd
                   For help type (help)

3.2
zal: (symbol-value 'x?)
5
zal: (1+ x?)
6
zal: (funcall #'(lambda (x?) (1+ x?)) 16)
17
zal: (symbol-value 'x?)
5
zal:
```

Figure 4-13 : Lambda variables are distinct from symbols of the same name

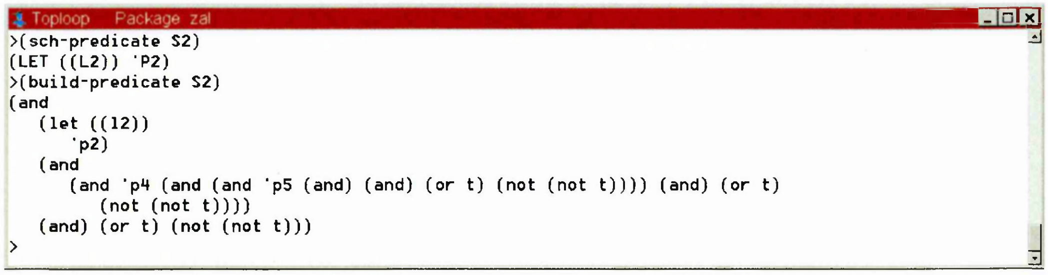
4.3.6.3.2 Output objects

Output objects in Z are suffixed with a '!' character and traditionally fulfil two functions: they are used to report outcomes, with a success message or an error report and they are used to pipe values between the two schemas of a composition, where the output of the first schema - say name! - is the input value -here name? – to the second. Though not usual, it is legal to treat an output object as a value that can be used in predicate clauses; consequently output objects, once instantiated, are manipulated as WORM[10]; they will generally be bound to a value at some point and may then be used as named values.

[10] Write Once Read Many

4.3.6.3.3 *Local objects*

Local objects are also exhibit WORM functionality, but their scoping is rather different from all the other classes. Whereas a declaration in one schema of the topology introduces an object to all the predicate clauses of the executable object, a local object is *de facto* local and is not visible in any other schemas in the topology. To implement this alternative scoping, a *let* expression is constructed when the schema object is defined, which declares a local object with the predicate of just the defining schema as its body. Again using the topology of , a local declaration of an object L2 in schema S2 will be reflected in the local and expanded predicate as in Figure 4-14


Toploop Package zal

```
>(sch-predicate S2)
(LET ((L2)) 'P2)
>(build-predicate S2)
(and
   (let ((l2))
      'p2)
   (and
      (and 'p4 (and (and 'p5 (and) (and) (or t) (not (not t)))) (and) (or t)
         (not (not t))))
   (and) (or t) (not (not t)))
>
```

Figure 4-14 : Local declaration incorporated into a schema predicate

In this situation, predicate P2 would reference the local object L2, which would not be visible to other elements of the expanded predicate.

4.3.6.3.4 *State objects*

What are described as state objects are those data components that are declared in the state schema declarations and which are constrained by the state schema predicate, or invariant. During an execution they also represent named values and are not updated except by the process of promotion; it could argued that the promotion only occurs *after* successful execution and that consequently is no more than an alternative, more convenient mechanism for the binding of new values, prior to a further execution.

4.3.6.4 Collecting inputs

The set of named inputs is generated as a by-product of the analysis of the namespace; each has already been identified as an input and categorised as such when the individual schema object is created. Each is presented to the user in a dialogue, see **Figure 5-13,** and a value is assigned to each by the user. These values, rather than the objects themselves, are constructed into a list that is the argument to the function application.

4.3.6.5 Evaluating the executable object

The key aspect, actually executing a schema, is simply a matter of applying the lambda expression that is the executable object to the actual parameters collected from the user. The environment of this application is that constructed from the declared objects. This application will yield a truth value, which is reported to the user.

4.3.6.6 Reporting the results of the evaluation

If the evaluation succeeded, any outputs are reported to the user, together with any other objects selected for display. Usually these will be state objects and in some executions, the post-state will differ from the pre-state. When this occurs, the user will be offered the option of promoting the post-state values, as in **Figure 5-5**.

Provided that the state schema was included in the schema that was executed, if all the post-state values are promoted then state data will remain self-consistent and valid; should some, but not all, of the values be promoted, this consistency is not guaranteed. However the invariant is executed automatically at the end of the promotion dialogue and should an inconsistent state exist, it is reported to the user, as demonstrated by Figure 5-6..

4.3.6.7 Management of the ongoing state

The state space under consideration here comprises the data objects declared in the state schema; these are the objects that represent the current state of the model and as such should be both updateable and persistent. They constitute the data that is global *to an animation* and are explicitly managed by the animation environment, primarily to ensure the self-consistency explicit in the state schema .

The current bindings for the state objects can be examined using the Binding Browser and these bindings can be modified by the user; because this introduces the possibility of an inconsistent state, the invariant is automatically executed after this editing, and violations reported.

The opportunity for establishing an inconsistent state is supported, since the reasoning with a specification may well include investigation of exactly that point – what constitutes an invalid state, with respect to any particular invariant.

Though it remains possible to access the command line, with the consequential possibility of a corrupted state, this is considered unlikely and would in any case be recognised by the mechanism described above.

4.4 The development of a developer's interface

To a great extent, the design of an integrated, cohesive interface is not feasible until it is known what functionality is to be interfaced; as perhaps the main focus of the development has been to investigate precisely that – what functionality can be implemented – then it is only now that the development of such an interface might reasonably be undertaken. However the inability to predict the scope of that functionality and how it might be used cannot preclude the provision of some mechanism to access it; consequently the interface as is implemented can best be thought of as a developer's interface, providing access to the functionality but with little attention having been paid to either its ergonomics or aesthetics. Similarly, the range of ways that the implemented functionality is used could not readily be predicted and so the intention has been to provide facilities that are both non-prescriptive and non-proscriptive in the way they are used.

While the actual appearance of the interface presented to a user might be considered *ad hoc* in style, it does remain simply an interface; all the functionality is available at the command line interpreter and more importantly within the ZAL language itself. For example, the ability to execute the invariant as discussed previously is accomplished by a simple function call to the execution function `execute`; this call can be made at the command line or it can be constructed interactively, together with its arguments, by the interface.

4.5 Optimisations

An investigative development such as this will not generally consider the optimisation of the performance as a concern; indeed this should not figure as a consideration of the specifier, who is free to use the full expressive power of the Z notation. Notwithstanding this, there may arise situations where the optimisation of *the transformed ZAL version* of a specification may render it feasibly executable whereas the un-optimised version is not.

Empirical evidence suggests that the mechanism for resolving the equality testing issue (see Section 4.3.4.2), the ZAL operator `eqz`, consumes significant resources of the platform on which the animation runs. The majority of individual predicate clauses use the equality operator and so it is a prime candidate for optimisation. Furthermore it is implemented as a macro which entails a large expansion whenever it is used, so some research has been undertaken to establish the feasibility of such optimisation. The strategy ultimately devised involves a series of optimisations, which are increasingly aggressive but decreasingly safe in their effect. These optimisations are managed as a user-configurable safety level, which controls which of the optimisations are invoked.

4.6 Other issues of implementation

Though not yet necessary, the ability to model non-determinism has been demonstrated through the provision of a non-deterministic choice operator, `choose`, after [Graham 1994], but this potential is not exploited in the system reported upon here, though it forms the basis of `non-deterministic-pick` – see Appendix E (choose); this was developed as an alternative to `pick,` as used in Section 5.3 (page 135), but its usefulness is limited and it remains of marginal relevance at this stage of the project

4.7 In conclusion

The realisation of the necessary animation functionality was an incremental exercise that involved only limited backtracking to correct inappropriate design decisions; the decision to delay the expansion and optimisation of the executable schema object –see Section *4.3.5*- was the only occasion which necessitated significant revision and the current realisation is a tribute to the effectiveness of exploratory programming as an approach to the resolution of ill-defined and ill-understood problems.

The functionality that has been achieved can now be examined through a number of case studies.

5 Case Studies

This section will explore the use of ZAL in a variety of problem domains and also in a variety of styles of use. The three studies are :

- a traditional data processing area, with a well-understood problem – that of a lending library. This has been chosen to demonstrate the capabilities and characteristics of the animation: in particular, an indication of the breadth and complexity of the Z expressions that can be executed, the support for elements of the schema calculus and the mechanisms for modelling generic and axiomatic definitions. The interaction might be typical of that involving the system specifier and the client, where the purpose would be to demonstrate the behaviour implicit in the specification – i.e. requirements validation;
- a safety-critical, monitoring context, involving a water-level monitoring system with its associated controls and alarms. The interaction demonstrated here is more typical of the developer of the specification exploring a complex scenario and using the animation to confirm that the recorded Z notation does in fact express the correct logic/behaviour – this would be a requirements formalisation process. This study also demonstrates the use of a scripting interface that could be developed to provide facilities to manage and administer test suites; and
- an analysis of a development in which the ZAL language is used as if it were a specification language – i.e. the specification is developed using the ZAL animator. Though this could be considered an inappropriate approach, given the expectation that the ZAL version has been generated from the Z, it will be shown to be consistent with the more usual usage. The classification of this study is less straightforward as it does not typify a Software Engineering activity *per se*; it originated as a genuine third-party undertaking, where the ZAL animator was being used in an unexpected way. That usage though is compatible with the stated objective "to make the specification more accessible", provided this includes the process of specification development; it also coincides with the contention that the use of the animator is not prescribed – the expectation is that it will be used within the REALiZE process, but other possibilities are not excluded.

Mention should be made of the source and status of the ZAL code that is being demonstrated, to clarify the purpose served by its presentation; rather than present just the final version of a development, some attention is paid to the process of refining the specification. As a consequence, successive versions of a schema may be introduced in support of some particular point and rather than confuse the picture with the detail of using TranZit to edit and transform, the Z version of some of these is omitted in the interests of clarity. In each case, the decision as to what to include has been governed by the need to expose the performance of the animator in a given situation, so that in the schemas suffixed '2', the ZAL code itself may have been edited.

The style of the animation in each case is interactive, which reflects the planned use of the tool. A consequence of this decision is that the scenarios that are explored in any animation are chosen by the user of the animation to confirm or explore some particular behaviour or situation; no mechanism currently exists for ensuring or managing more exhaustive testing.

5.1 A Lending Library

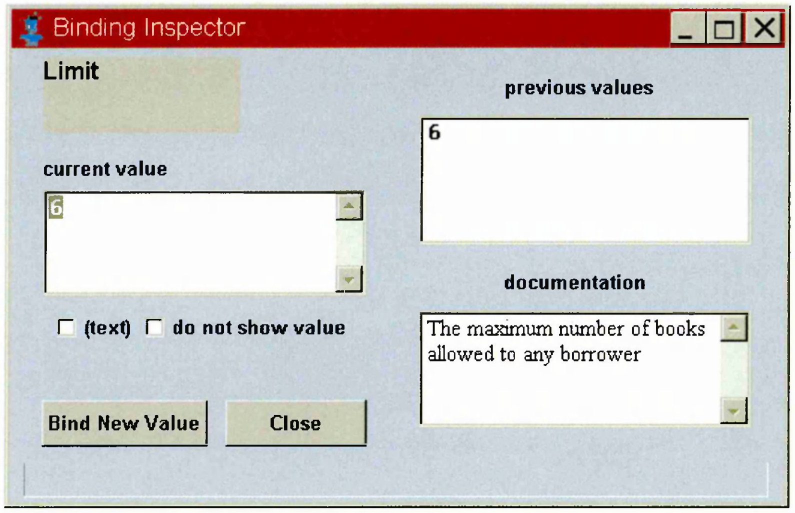
The library system examined here derives originally from [Diller 1990] and models a lending library which records the location and status of the texts in stock and also the members of the library – the borrowers. The primary purpose is to demonstrate the ZAL animator operating in "normal" mode – that is, demonstrating the functionality of the specification to the sponsor actor to confirm that the behaviour as formally recorded is consistent with user intention.

The original Z is much the best description of the behaviour we seek to model; it is reproduced here, together with rather extensive comments, and contrasted with the ZAL code that models it.

```
[BOOK, COPY, PERSON]
```

The given sets of a specification are populated as required; type-checking would be their principal use in the animator, but that function is achieved by TranZit. Consequently it is sufficient to instantiate values to the data components of (usually) the state schema. Here it is also necessary to bind the constant `limit` to the value 6. This can be done with a ZAL script or interactively using the tools of the ZAL Developer Interface.

$$
\begin{array}{|l}
\text{limit} : \mathbb{N} \\
\hline
\text{limit} = 6
\end{array}
$$



current value: 6

previous values: 6

documentation: The maximum number of books allowed to any borrower

Figure 5-1 : The Binding Inspector

In fact, values have been bound to each of the data components of the state schema Library, and these can be browsed and edited using the Binding Browser and Binding Inspector; they can also be echoed at the ZAL command line, thus

```
zal: stock
{'C1 'C2 'C3 'C4 'C5 'C6 'C7 'C8}
zal: shelved
{'C1 'C2 'C3}
zal: borrowed
{'C4 'C5 'C8}
zal: loans
[#('C4 'BOB) #('C5 'ANN) #('C8 'IAN)]
zal: members
{'ANN 'BOB 'IAN 'TED 'TOM}
zal: keptaside
{'C6 'C7}
zal: keptasidefor
[#('C6 'CAROL) #('C7 'PAUL)]
zal: isacopyof
[#('C1 'CATCH22) #('C2 'SPOT) #('C3 'CATCH22) #('C4 'BLEAKHOUSE) #('C5 'SPOT) #('C6
'COCKATOOS) #('C7 'THOMASTHETANKENGINE) #('C8 'SPOT)]
zal: hasreserved
[#('BOB 'SPOT) #('TED 'BLEAKHOUSE) #('TED 'SPOT) #('TOM 'BLEAKHOUSE)]
zal:
```

These values are a self-consistent set of data that satisfies the system invariant as represented by the predicate of the state schema.

The function `numberBorrowedBy` is defined in Z axiomatically, using a quantification; the quantification serves to declare the domain of the function. The ZAL code reflects the totality of the function by *not* constraining the argument to membership of some set of values; this construct can be dispensed with when modelled by a more natural function definition. It should be noted however that the precision and clarity of the Z definition are captured in the equivalent, so that

$\forall p : PERSON \bullet NumberBorrowedBy\ p = \#((loans^{\sim}) \llparenthesis \{p\} \rrparenthesis)$

becomes

```
(defun NumberBorrowedBy (p)
       (card ( rel-image {p} (inverse loans) )))
```

The state schema for the library can now be defined.

Library

$\text{stock, shelved, borrowed, keptAside} : \mathbb{P}\ \text{BOOK}$

$\text{members} : \mathbb{P}\ \text{PERSON}$

$\text{loans} : \text{COPY} \nrightarrow \text{PERSON}$

$\text{isACopyOf} : \text{COPY} \rightarrow \text{BOOK}$

$\text{hasReserved} : \text{PERSON} \leftrightarrow \text{BOOK}$

$\text{keptAsideFor} : \text{COPY} \nrightarrow \text{PERSON}$

$\text{stock} = \text{shelved} \cup \text{borrowed} \cup \text{keptAside}$

$\text{disjoint} \langle \text{shelved, borrowed, keptAside} \rangle$

$\text{borrowed} = \text{dom loans}$

$\text{keptAside} = \text{dom keptAsideFor}$

$\text{ran loans} \subseteq \text{members}$

$\text{dom hasReserved} \subseteq \text{members}$

$\text{dom isACopyOf} = \text{stock}$

$\forall\ m : \text{members} \bullet \text{NumberBorrowedBy}\ m \leq \text{limit}$

The correspondence of the ZAL version of this schema can be readily recognised; each predicate of the Z version has a matching predicate in ZAL, but in the prefix notation perhaps more familiar to Lisp users, thus

```
(SCHEMA Library
 :PREDICATE
 (and
      (eqz stock (unionz (unionz shelved borrowed )KeptAside ))
      (disjoint <shelved borrowed KeptAside>)
      (eqz borrowed (dom loans) )
      (eqz KeptAside (dom KeptAsideFor) )
      (subset (ran loans) members )
      (subset (dom HasReserved) members )
      (eqz (dom IsACopyOf) stock )
      (forall m members
       (\<= (NumberBorrowedBy m) limit ))
      (not-mem nobody members )
      )
 )
```

The self-consistency can be demonstrated by the original technique of executing the state schema using the Execution Tool – see Figure 5-2 : Executing the State Schema; this involves selecting a particular schema (here Library) and clicking the Run button

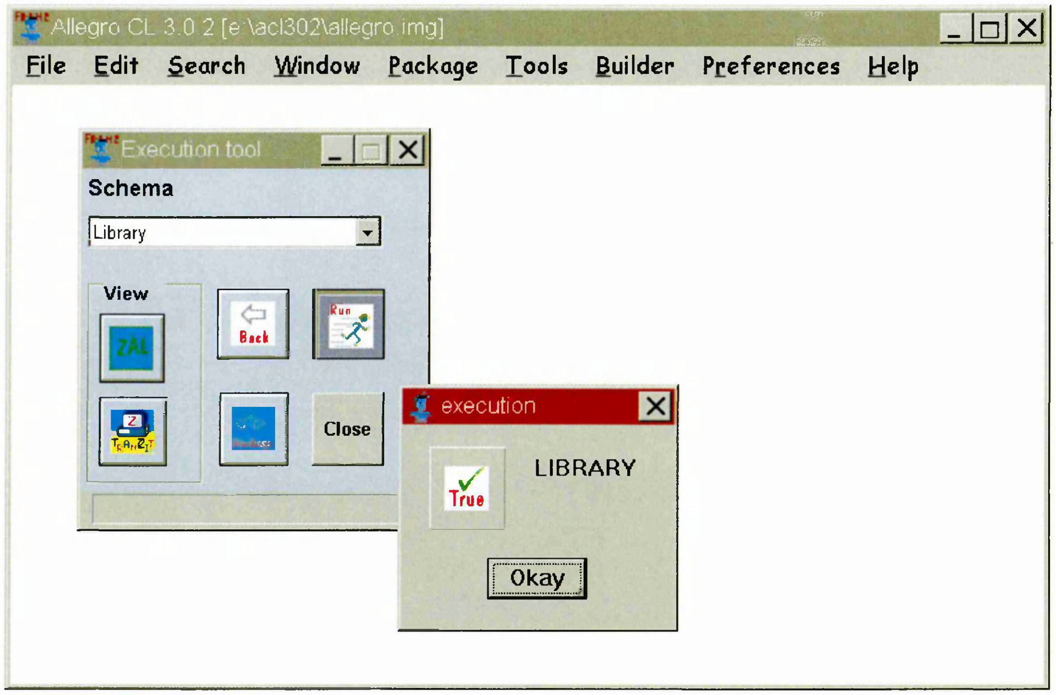


Figure 5-2 : Executing the State Schema

We will consider borrowing and returning, as well as querying the system. Borrowing a book, or more precisely a copy of a book, is straightforward; the copy is either shelved or has been kept aside for that borrower in response to a reservation request. In both cases the sets are "updated" using the set difference operation.

BorrowBook

$\Delta Library$

$c? : COPY$

$p? : PERSON$

$p? \in members$

$NumberBorrowedBy\ p? < limit$

$(c? \mapsto p? \in KeptAsideFor \vee c? \in shelved)$

$loans' = loans \cup \{ c? \mapsto p?\}$

$HasReserved' = HasReserved$

$KeptAsideFor' = KeptAsideFor \setminus \{c? \mapsto p?\}$

$shelved' = shelved \setminus \{c?\}$

This is modelled by the ZAL schema

```
(SCHEMA BorrowBook
   :INCLUDE delta-Library
   :SHOW (loans KeptAsideFor Shelved)
   :? (c? p?)
   :PREDICATE
   (and
      (mem p? members )
      (\< (NumberBorrowedBy p?) limit )
      (or
         (mem #( c? p? ) KeptAsideFor )
         (mem c? shelved )
      )
      (eqz loans' (unionz loans  [#( c? p? ) ] ))
      (eqz HasReserved' HasReserved )
      (eqz KeptAsideFor' (setsub KeptAsideFor  {#( c? p? ) }))
      (eqz shelved' (setsub shelved {c?}))
   )
)
```

There are a number of components of this operation schema that were not found in the state schema, which consisted solely of a `:PREDICATE` declaration; the `:INCLUDE` tag is used to support schema inclusion, in this case the ΔLibrary schema; the `:?` tag identifies the inputs to the schema, here `c?` and `p?` for the copy and borrower respectively; and the `:SHOW` tag controls the feedback to the user after the execution.

Inputs are collated interactively as the schema is executed, as can be seen from Figure 5-3 : Collecting Inputs.

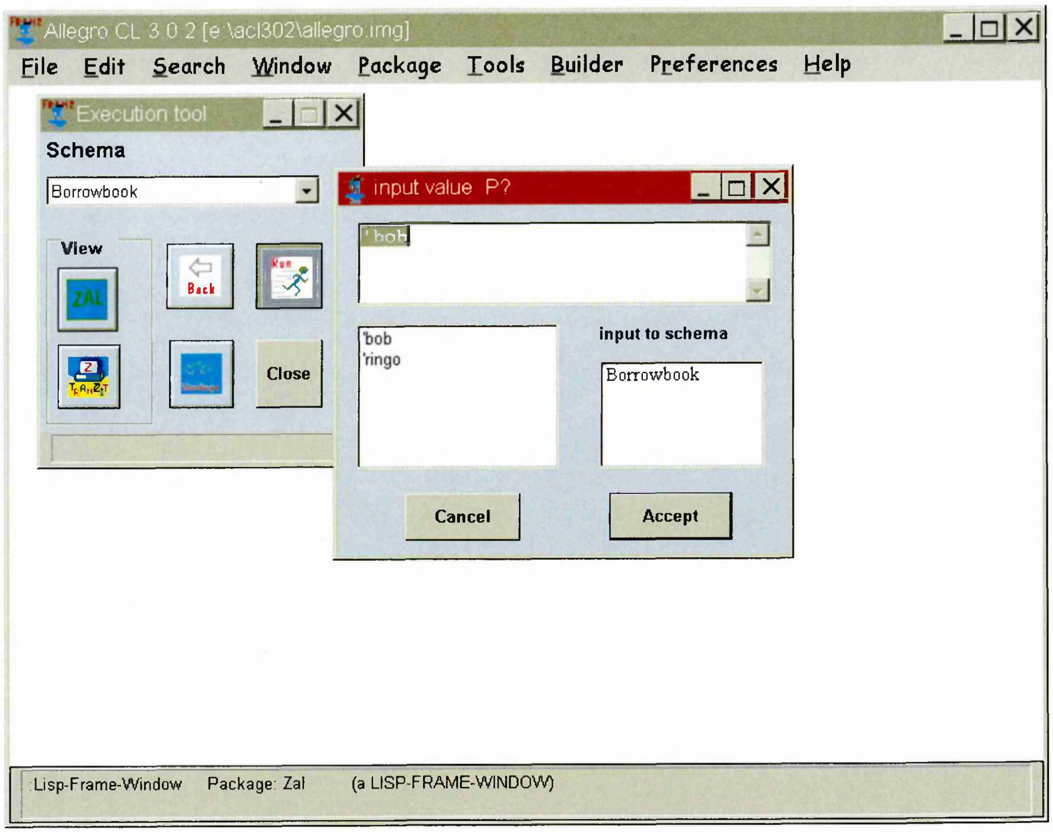
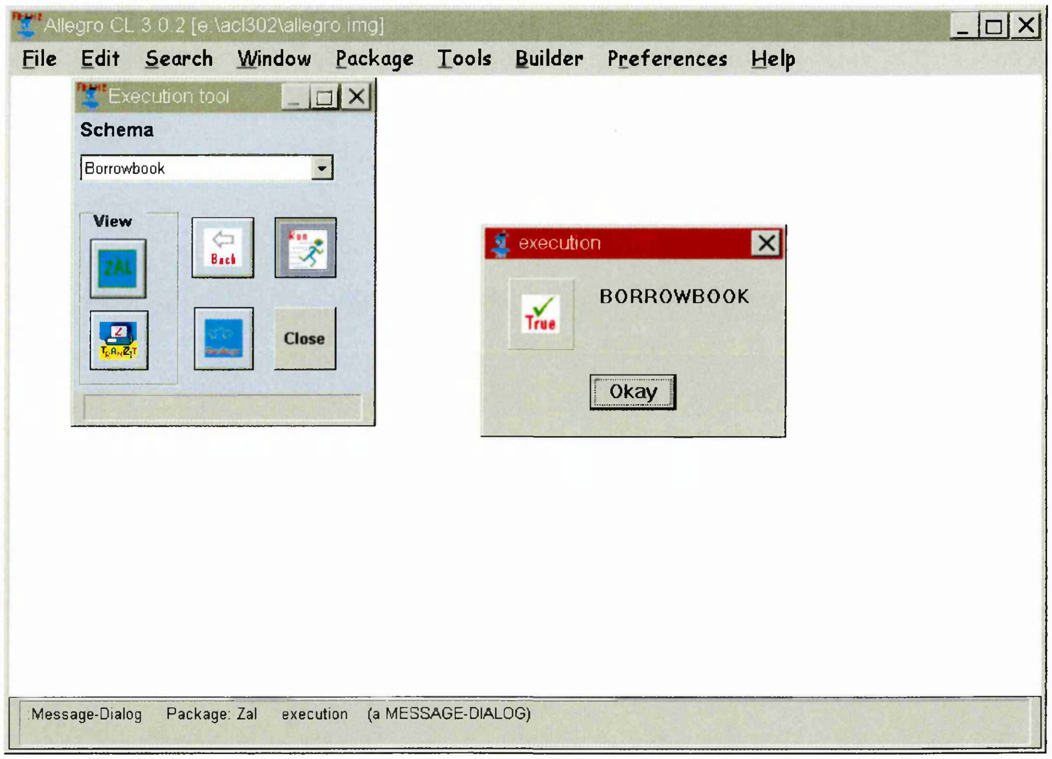

Allegro CL 3.0.2 [e:\acl302\allegro.img]
File Edit Search Window Package Tools Builder Preferences Help
Execution tool
Schema
Borrowbook
View
Back
Close
input value P?
'bob
'bob
'ringo
input to schema
Borrowbook
Cancel
Accept
Lisp-Frame-Window Package: Zal (a LISP-FRAME-WINDOW)

Figure 5-3 : Collecting Inputs

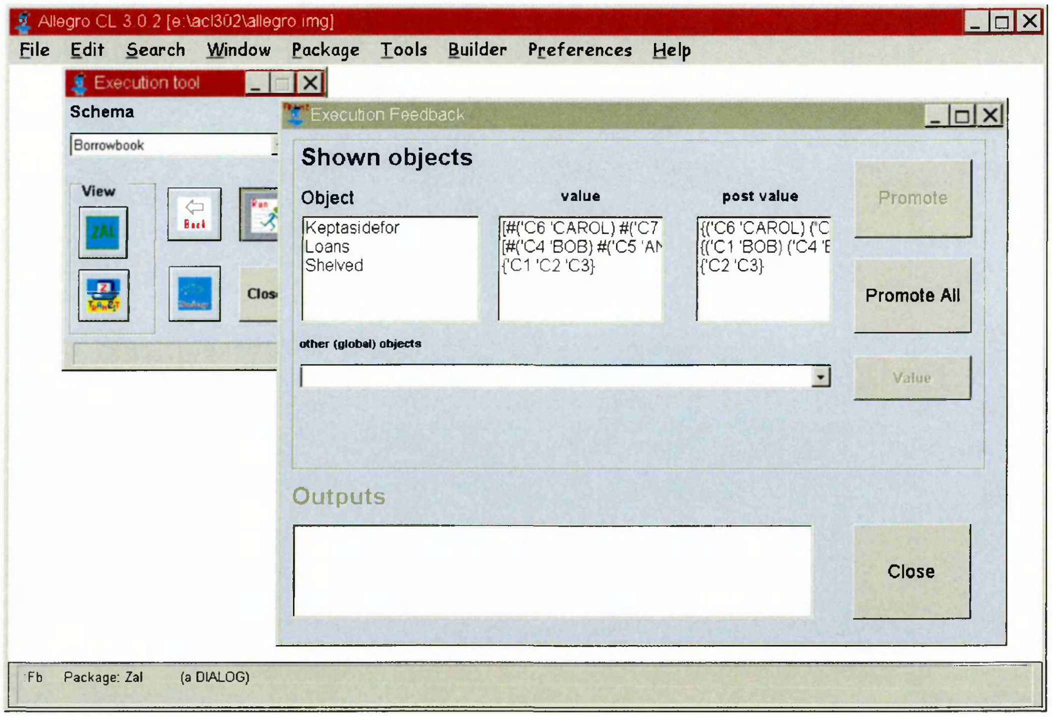
The successful execution –see Figure 5-4 - returns True and the values of the data objects are displayed -Figure 5-5; there is the opportunity after a successful execution to promote the decorated values and thereby to model a sequence of actions in a style that mimics somewhat schema composition.


Allegro CL 3.0.2 [e:\acl302\allegro.img]
File Edit Search Window Package Tools Builder Preferences Help
Execution tool
Schema
Borrowbook
View
Back
Close
execution
True
Okay

BORROWBOOK

Message-Dialog Package: Zal execution (a MESSAGE-DIALOG)

Figure 5-4 : Successful Execution


Allegro CL 3.0.2 [e:\acl302\allegro.img]
File Edit Search Window Package Tools Builder Preferences Help
Execution tool
Schema
View
Back
Close
Execution Feedback
Shown objects
Promote
Promote All
Value
Outputs
Close

Schema: Borrowbook

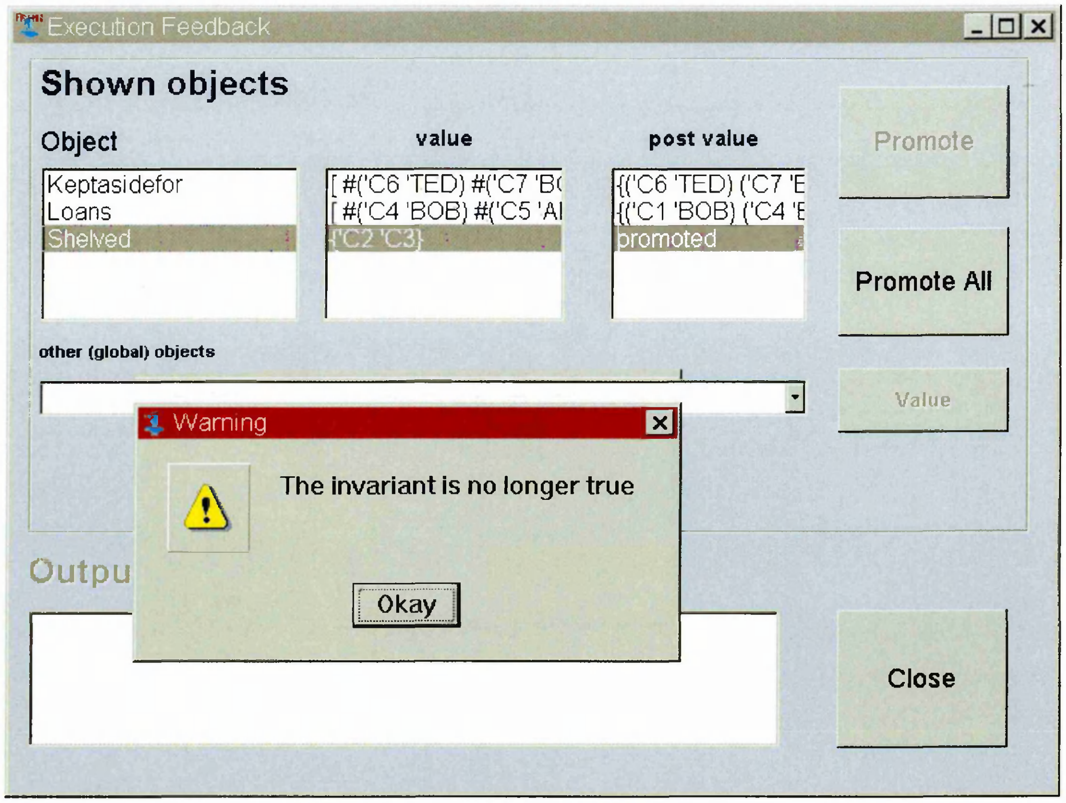
Object	value	post value
Keptasidefor	[#('C6 'CAROL) #('C7	{('C6 'CAROL) ('C
Loans	[#('C4 'BOB) #('C5 'AN	{('C1 'BOB) ('C4 'E
Shelved	{'C1 'C2 'C3}	{'C2 'C3}

other (global) objects

```
Fb    Package: Zal        (a DIALOG)
```

Figure 5-5 : Execution Feedback

Promotion of all the post-state values will ensure a consistent state, provided that the state schema has been included, the predicate of which validates the state objects; if not all the post-state values are promoted, the possibility of an invalid state exists. This can be identified by the *automatic* execution of the state schema to provide that validation, as shown in Figure 5-6.


Execution Feedback

Shown objects

Object	value	post value
Keptasidefor	[#('C6 'TED) #('C7 'B(	{('C6 'TED) ('C7 'E
Loans	[#('C4 'BOB) #('C5 'AI	{('C1 'BOB) ('C4 'E
Shelved	{'C2 'C3}	promoted

other (global) objects

Warning

The invariant is no longer true

Figure 5-6 : Automatic warning of state inconsistency, after partial promotion

Returning a copy of a book also has two situations to consider. If a book has been reserved, when a copy of that book is returned, it is kept aside for the borrower who requested it and that particular reservation request is deleted. If it is not reserved, the copy is shelved once again. Regardless of reservations, the loans function must be updated.

Depending on the style of the Z used to describe this operation, at least two approaches are possible to identify the particular borrower for whom this copy will be kept aside; the Z below uses an existential quantification to both establish if the returned text is a copy of a reserved book and then to select a borrower from the domain of HasReserved

```
┌ReturnBook──────────────────────────────────
│ ΔLibrary
│ c? : COPY
│ p? : PERSON
├────────────────────────────────
│ c? ↦ p? ∈ loans
│ loans' = loans \ {c? ↦ p?}
│ (∃ m : members ; b : BOOK •
│         IsACopyOf c? = b ∧
│         HasReserved m = b ∧
│         KeptAsideFor' = KeptAsideFor ∪ {c? ↦ m} ∧
│         shelved' = shelved ∧
│         HasReserved' = HasReserved \ {m ↦ b}
│ )
│ ∨
│ (¬ ∃ m : members ; b : BOOK •
│         IsACopyOf c? = b ∧
│         HasReserved m = b ∧
│         KeptAsideFor' = KeptAsideFor ∧
│         shelved' = shelved ∪ {c?} ∧
│         HasReserved' = HasReserved
│ )
└──────────────────────────────────────
```

This translates and executes as

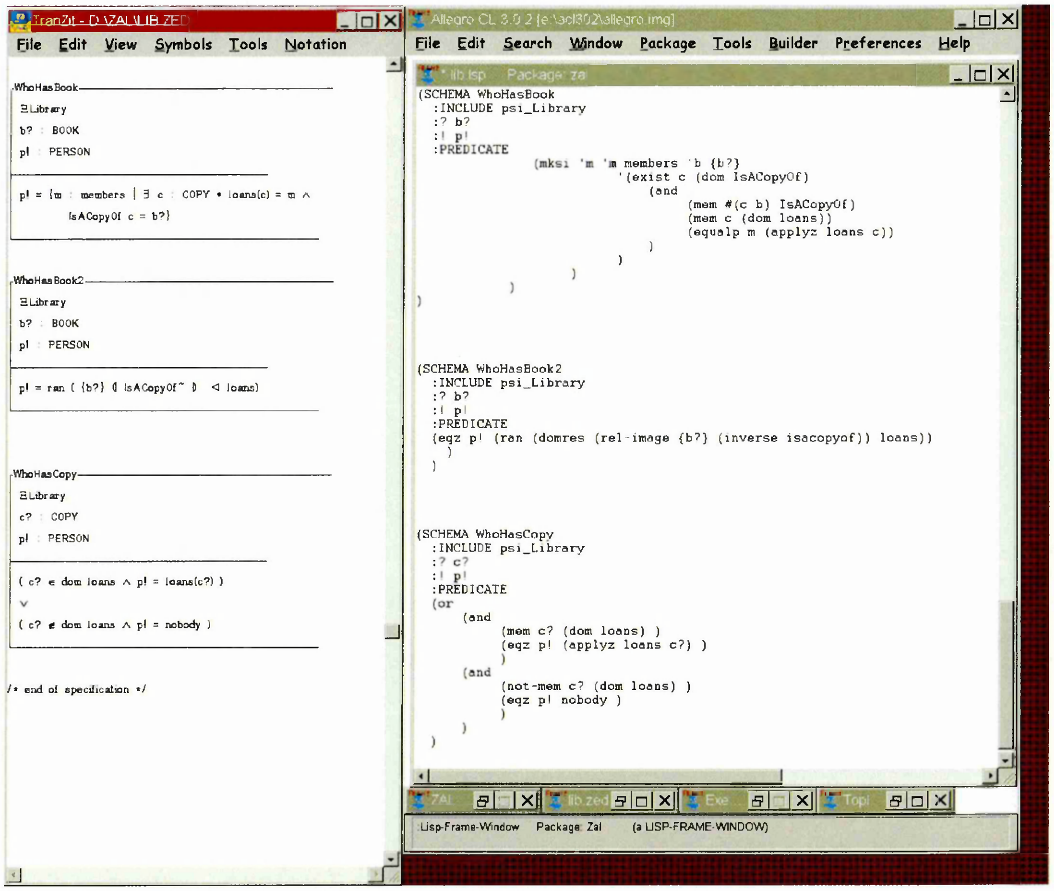
```
(SCHEMA ReturnBook
 :INCLUDE delta-Library
 :SHOW (loans KeptAsideFor shelved HasReserved )
 :? (c? p?)
 :PREDICATE
 (and
     (mem #( c? p? ) loans )
     (eqz loans' (setsub loans  {#( c? p? ) }))
     (or
         (and
              (exist m members b (ran IsACopyOf)
               (and
                    (equalp (title c?) b )
                    (mem #( m b ) HasReserved)
                    (eqz KeptAsideFor' (unionz KeptAsideFor  {#( c?  m)} ))
                   (eqz shelved' shelved )
                   (eqz HasReserved' (setsub HasReserved {#(m (title c?))} )))
         (and
              ( not
                (exist m members b (ran IsACopyOf)
                 (and
                      (eqz (applyz IsACopyOf c?) b )
                      (mem #( m b ) HasReserved))
                      (eqz KeptAsideFor' KeptAsideFor )
                     (eqz shelved' (unionz shelved  {c?  }))
                     (eqz HasReserved' HasReserved ) ))
              )
         )
     )
```

The non-determinism of the Z, whereby it is not specified for which reserving borrower the returned book is kept aside, is not modelled in the ZAL translation; the animation will in fact allocate the returned book to the alphabetically first member that has reserved a copy and the updating predicates for subsequent 'valid' reservers that are tested will fail, as the post-states have already been bound to reflect the success of the first valid reserver. Somewhat cleaner is to check for the book title as a member of the range of the reserved books; this test is very much clearer, but the mechanism for selecting the "winning" reserver is much less obvious. A valid approach is to choose the 'first' reserver according to some criterion

```
(SCHEMA ReturnBook2
 :INCLUDE delta-Library
 :SHOW (loans KeptAsideFor shelved HasReserved )
 :? (c? p?)
 :LOCALS (winner book-name)
 :PREDICATE
 (and
      (eqz book-name (title c?))
      (eqz winner (1st-reserver book-name))
      (mem #( c? p? ) loans )
      (eqz loans' (setsub loans  {#( c? p? ) }))
      (or
          (and
               (mem book-name (ran HasReserved))
               (eqz KeptAsideFor' (unionz KeptAsideFor  {#( c?  winner)} ))
               (eqz shelved' shelved )
               (eqz HasReserved' (setsub HasReserved  {#(winner book-name)} ))
               )
          (and (not-mem book-name (ran Hasreserved))
               (eqz KeptAsideFor' KeptAsideFor )
               (eqz shelved' (unionz shelved  {c?  }))
               (eqz HasReserved' HasReserved )
               )
          )
      )
 )
```

This schema introduces another tag :LOCALS which provides for the introduction of local objects. The detail of the choice of reserver is now abstracted into the function 1st-reserver; this abstraction will allow more sophisticated models of reservation, probably involving a sequence of reservers, to be introduced.

The Z and ZAL versions of the two query schemas**WhoHasBook** and **WhoHasCopy** are presented as a screen captured from TranZit and ZAL running in individual windows (Figure 5-7).


TranZit - D:\ZAL\LIB.ZED
File Edit View Symbols Tools Notation
Allegro CL 3.0.2 [e:\acl302\allegro.img]
File Edit Search Window Package Tools Builder Preferences Help
*lib.lsp Package: zal
ZAL
lib.zed
Exe
Topl

```
WhoHasBook
ΞLibrary
b? : BOOK
p! : PERSON
─────
p! = {m : members | ∃ c : COPY • loans(c) = m ∧
        IsACopyOf c = b?}

WhoHasBook2
ΞLibrary
b? : BOOK
p! : PERSON
─────
p! = ran ( {b?} ⦇ IsACopyOf~ ⦈  ◁ loans)

WhoHasCopy
ΞLibrary
c? : COPY
p! : PERSON
─────
( c? ∈ dom loans ∧ p! = loans(c?) )
∨
( c? ∉ dom loans ∧ p! = nobody )

/* end of specification */
```

```
(SCHEMA WhoHasBook
  :INCLUDE psi_Library
  :? b?
  :! p!
  :PREDICATE
                 (mksi 'm 'm members 'b {b?}
                            '(exist c (dom IsACopyOf)
                                (and
                                     (mem #(c b) IsACopyOf)
                                     (mem c (dom loans))
                                     (equalp m (applyz loans c))
                                )
                            )
                      )
             )
)


(SCHEMA WhoHasBook2
  :INCLUDE psi_Library
  :? b?
  :! p!
  :PREDICATE
  (eqz p! (ran (domres (rel-image {b?} (inverse isacopyof)) loans))
    )
  )


(SCHEMA WhoHasCopy
  :INCLUDE psi_Library
  :? c?
  :! p!
  :PREDICATE
  (or
      (and
           (mem c? (dom loans) )
           (eqz p! (applyz loans c?) )
           )
      (and
           (not-mem c? (dom loans) )
           (eqz p! nobody )
           )
      )
  )
```

Lisp-Frame-Window   Package: Zal   (a LISP-FRAME-WINDOW)

Figure 5-7 : The Toolset Components in Parallel

There are two versions of the schema WhoHasBook; the first uses a set comprehension to identify borrowers of a particular title and the second utilises function inversion, relational image and domain restriction to achieve the required functionality. This contrast in styles is examined in a little more detail in Section 5.3 .

It can be seen from this exploration that the ZAL engine can indeed animate a specification in the conventional 'data-processing' area; this is certainly to be expected, but the ability to execute quite complex Z expressions, such as

$$\begin{array}{l} \exists\ m : members\ ;\ b : BOOK \bullet \\ \qquad IsACopyOf\ c? = b \wedge \\ \qquad HasReserved\ m = b \wedge \\ \qquad KeptAsideFor' = KeptAsideFor \cup \{c? \mapsto m\} \wedge \\ \qquad shelved' = shelved \wedge \\ \qquad HasReserved' = HasReserved \setminus \{m \mapsto b\} \end{array}$$

or

$$p! = ran\ (\ \{b?\}\ (\!|\ isACopyOf^{\sim}\ |\!)\ \lhd\ loans\ )$$

is perhaps noteworthy.

5.2 The Water Level Monitoring System

Introduction

The second case study concerns requirements validation in the domain of high integrity, safety-critical systems, where the use of formal methods has been most widely adopted. A Z version is produced of a published VDM specification for a water-level monitoring system (WLMS), which is then reasoned with, by inspection but more rigorously by animation; this exercise was part of a larger enterprise, which reports upon the REALiZE method referred to in Section **3.4**. The safety properties of the system are validated by animation. In particular, this example illustrates its use in conjunction with another component of the toolset, namely TranZit.

The principal area of interest here is the automatic production of the ZAL code and its execution, but a brief reprise of the REALiZE method and the toolset may be helpful to contextualise the particular process on which we are focusing.

The REALiZE method is a framework for the interaction between requirements acquisition, requirements formalisation and requirements validation, and as a protocol for the integrated use of the TranZit and ZAL toolset components; it is located in the standard software lifecycle model at the requirements analysis phase. After the initial requirements capture, the specifier formalises the requirements in the Z notation using the facilities provided by the TranZit tool. Once this is complete, the specifier can validate these requirements. The specifier uses the TranZit tool to produce an executable representation of the captured Z specification in the ZAL language. This representation can then be executed by the specifier within the ZAL animation environment, for the purposes of demonstrating properties of the captured specification to members of the stakeholder team.

5.2.1 The Z specification used in the case study

The WLMS is a typical, though small-scale, safety system, in that it is a real-time event-driven system. [Jackson and Stokes1993] specified the system using VDM and implemented it in Pascal. For this case study it was translated into Z. It was decided to translate the parts of the system which monitor inputs and use the readings to specify the state of the pump switch and an alarm signal, but to ignore (for simplicity) the parts of the system which are concerned with display of the state on a monitor. In its overall structure, the VDM specification and its Pascal implementation consist of an initialisation operation and a main operation which is run repeatedly. The main operation takes in inputs from a clock, water-level sensor, control buttons, etc., and specifies values for state variables and outputs such as a switch to control the pumps and an alarm which gives an audible warning when problems occur in the system. The main operation uses a number of operations which deal with parts of the system, and in some cases these are subdivided further. The system can be in one of four main (operating) modes. It begins in standby mode; then if a reset button is held down for a certain time and the water level is within its correct limits, it changes to operating mode, with the pumps running. If the water level is found during operating mode to be outside the correct limits, it changes to shutdown mode. If the water level then recovers within a certain time, the pumps do not stop and it returns to operating mode; if not, then the pumps are switched off and standby mode is entered. There is a test mode which is entered from any other mode by holding down a test button. Another state variable records the failure mode. This is normally `allok`, but becomes `badlevdev` if the device for monitoring water level fails, or `hardfail` if other hardware fails.

The production of the specification was strictly a translation process from VDM to Z which demonstrated some differences in the styles of specification encouraged by the two languages. The basic types and constants in the VDM specification were translated in a straightforward manner to free types and global variables in Z. However, the structured types in the VDM were translated simply using inclusion of state schemas. Real numbers are used in the VDM specification for water levels and times. As mentioned by [Jackson and Stokes 1993], floating point implementations of real numbers present problems for formal verification. For this reason, the basic Z language and toolkit [Spivey 1992] did not originally include real numbers, though these have been added by [Valentine 1993], and integer types were used in the Z specification here. In practice, a time is a number of clock ticks, and the water level in this system is derived from a differential pressure which can take only 256 discrete values, so the use of integers is quite natural. The specification was also simplified by putting many small schemas into a few large ones, and by avoiding the use of structured types (implemented by schema inclusion) in the state. The complete resulting specification is shown in Appendix C (WLMS in Z), though much of it is also included here.

A number of errors were identified during the process of simplification; these had to be corrected to make the simplified structure possible and to allow animation. An error in the operation to set a new state when inputs are read by the monitoring system in standby mode was noticed by [Jackson and Stokes 1993] if buttons were pressed and released at certain times, the operation required the operating mode variable to have two different values. This type of error is not possible in ``if-then-else" style of Z which was adopted to translate the implication connectives used in the VDM to specify the values of variables, because the conditions under which different values are specified are orthogonal under this style of Z. The operation which determines the state of the alarm (silent or audible) in the original specification contained a precondition which required that the power was on and the hardware had not failed. This was not checked when the operation was included in the main operation function (VDM) or schema (Z). This results in an operation with a predicate which cannot be satisfied by any values of the state variables when the precondition fails. In the simplified version, a third value `undefined` was introduced to allow animation to continue when the precondition fails. In the original specification, the state variable waterlevel was not uniquely determined from the input pressure diffPress? if the latter had one of its extreme values 0 or 255 which should never occur in normal operation. This non-determinism was not serious because the range of allowable values caused no problems in the rest of the specification.

A number of misprints were introduced in the translation from VDM to Z. A ' ¬ ' symbol was omitted before the test to check that the water level was correct in the schema which sets the new operating mode. This was noticed when the Z was translated to ZAL, but was left in to demonstrate that it could be detected by the animation. The water level was not determined correctly from the input pressure because of a missing zero in a number used in the calculation. This error was also not corrected.

5.2.2 Validation

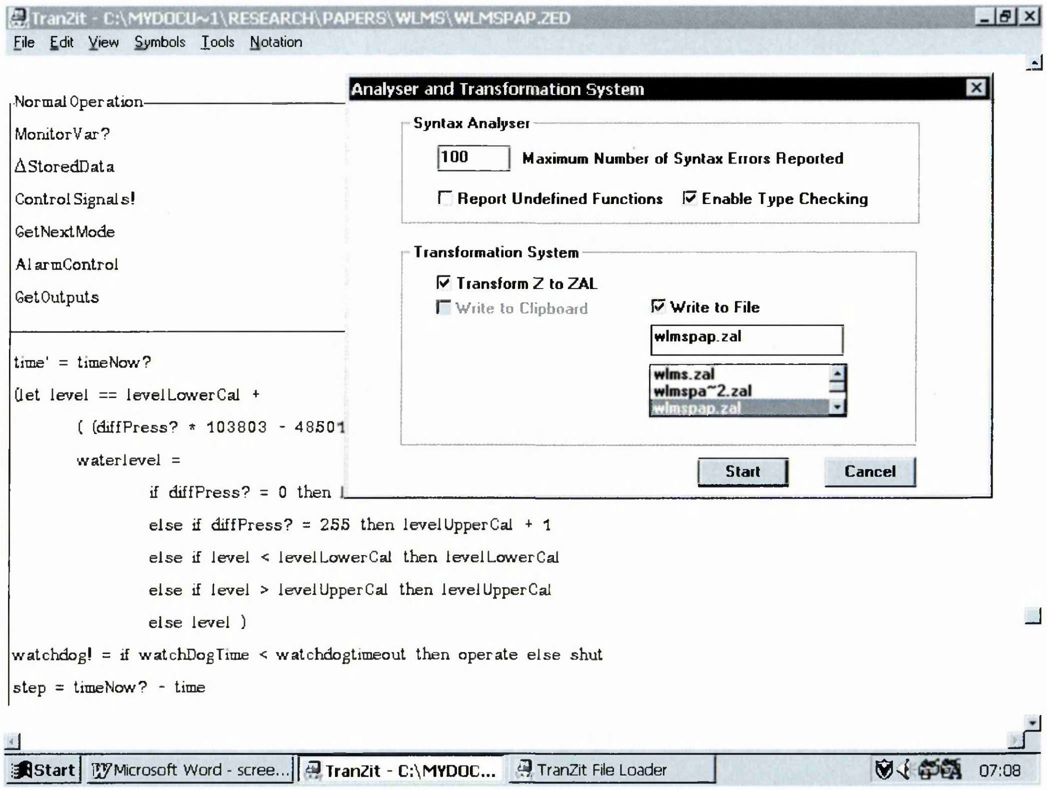
Before detailing the animation of the specification, brief consideration is given to the issues involved in animating specifications for safety-critical applications. Whatever checks and formal static analysis have been carried out on safety-critical software, dynamic testing of the system as implemented is essential to produce evidence that the software causes the target computer and plant to behave as intended. Testing of safety-critical software uses the same techniques as for non-critical systems, but the whole process is performed within a more rigorous, quality-assured framework. This comprises part of the safety case for the system. This requires that the tests performed, the inputs and output results, are all documented and placed under change control. Test definitions and coverage analysis are required, for example, by DO--178A for critical and essential functions. This means checking the test cases to discover which aspects of the requirements are confirmed by successful execution of each test [Pile 1991].

However, animation is not implementation, and its purpose is both to find errors in the formal specification and to identify problems in the informal requirements. Validation at the requirements phase of safety-critical software is much less well understood than the testing of implementations, and different approaches are still under development. There is as yet no standard, agreed method, but usually a collection of techniques borrowed from other kinds of safety-critical engineering is employed. Test cases for the requirements phase would be generated using standard safety analysis techniques such as: Event-tree analysis; Fault-tree analysis; Failure mode effects analysis (FMEA), Failure mode effect and criticality analysis (FMECA); Hazard and operability studies (HAZOP) [Pile 1991]. Given the constraints of the exercise being described, significantly fewer, if any, iterations through the cycle and versions of the specification would be expected and very little change to the specification is reported here, beyond the identification and correction of errors, including those described earlier.

5.2.3 Animation

There follows a demonstration of an animation that was used to investigate the behaviour of the system as specified; the brief commentary guides the reader through the process.

Once the complete specification of the WLMS has been syntax- and type-checked by TranZit, the transformation engine can be used to produce the ZAL executable version (Figure 5-8)


TranZit - C:\MYDOCU~1\RESEARCH\PAPERS\WLMS\WLMSPAP.ZED
File Edit View Symbols Tools Notation
Analyser and Transformation System
Start
Cancel
Start
Microsoft Word - scree...
TranZit - C:\MYDOC...
TranZit File Loader
07:08

```
Normal Operation
MonitorVar?
ΔStoredData
ControlSignals!
GetNextMode
AlarmControl
GetOutputs

time' = timeNow?
(let level == levelLowerCal +
        ( (diffPress? * 103803 - 48501
        waterlevel =
                if diffPress? = 0 then l
                else if diffPress? = 255 then levelUpperCal + 1
                else if level < levelLowerCal then levelLowerCal
                else if level > levelUpperCal then levelUpperCal
                else level )
watchdog! = if watchDogTime < watchdogtimeout then operate else shut
step = timeNow? - time
```

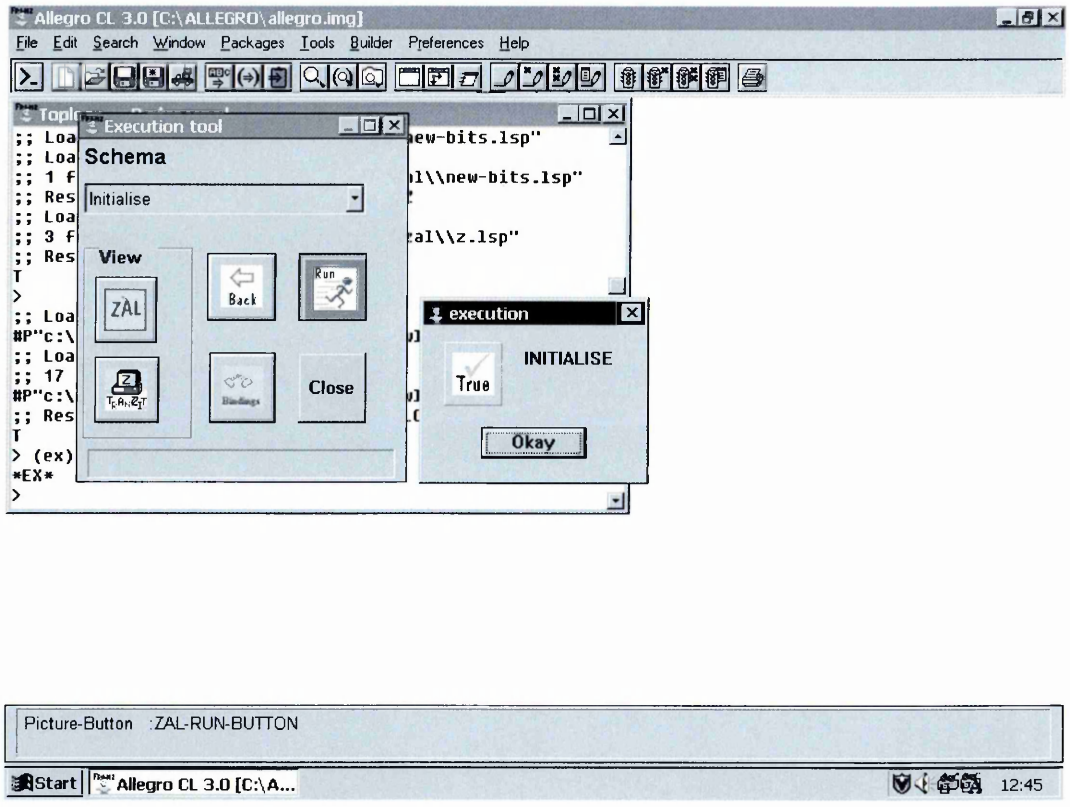
Syntax Analyser
- 100 Maximum Number of Syntax Errors Reported
- [ ] Report Undefined Functions
- [✓] Enable Type Checking

Transformation System
- [✓] Transform Z to ZAL
- [ ] Write to Clipboard
- [✓] Write to File: wlmspap.zal
- wlms.zal
- wlmspa~2.zal
- [✓ wlmspap.zal]

Figure 5-8 : The Transformation Process

This can then be used by the ZAL system to interactively investigate the normal operation of the WLMS, as specified in the schema NormalOperation

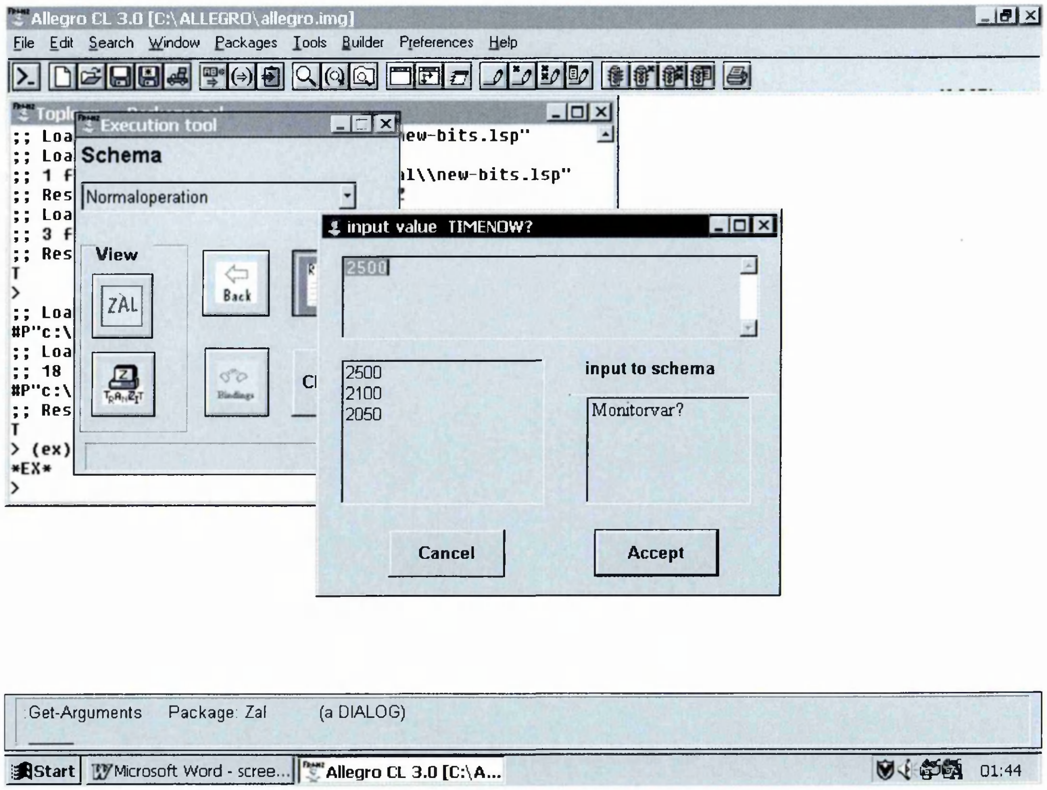
On entering the ZAL environment and loading the executable version of the specification, the ``Execution Tool" is invoked (Figure 5-9).



```
;; Loa                ew-bits.lsp"
;; Loa                l\\new-bits.lsp"
;; 1 f
;; Res
;; Loa
;; 3 f                al\\z.lsp"
;; Res
T
>
;; Loa
#P"c:\
;; Loa
;; 17
#P"c:\
;; Res
T
> (ex)
*EX*
>
```

```
Picture-Button   :ZAL-RUN-BUTTON
```

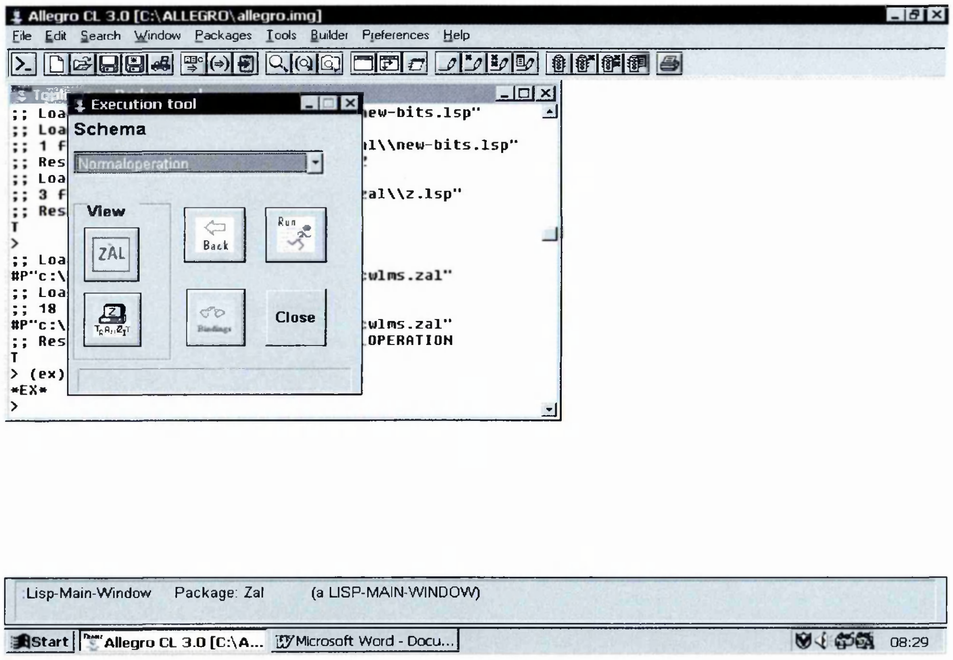
Figure 5-9 : The Execution Tool


Allegro CL 3.0 [C:\ALLEGRO\allegro.img]
File Edit Search Window Packages Tools Builder Preferences Help
Execution tool
Schema
Normaloperation
View
Back
input value TIMENOW?
2500
2500
2100
2050
input to schema
Monitorvar?
Cancel
Accept
Get-Arguments Package: Zal (a DIALOG)
Start
Microsoft Word - scree...
Allegro CL 3.0 [C:\A...
01:44

```
;; Loa        ew-bits.lsp"
;; Loa
;; 1 F        l\\new-bits.lsp"
;; Res
;; Loa
;; 3 F
;; Res
T
>
;; Loa
#P"c:\
;; Loa
;; 18
#P"c:\
;; Res
T
> (ex)
*EX*
>
```

Figure 5-10 : Default input values

First, the Initialise schema is selected in the drop-down schema menu, and the schema is executed, by clicking the Run button, to give initial values to the variables alarm, shutdownSignal , timeInMode etc. The executed predicate evaluates to True. Then the NormalOperation schema is selected (Figure 5-11).

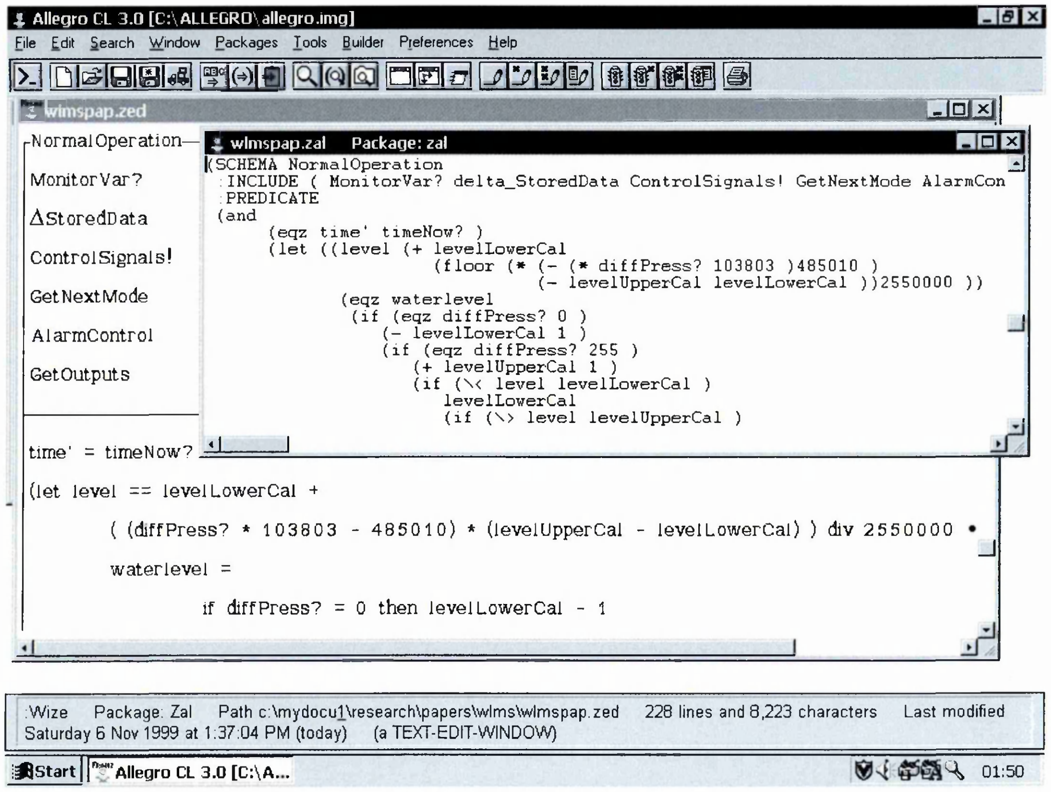


```
;; Loa                    ew-bits.lsp"
;; Loa
;; 1 f                    l\\new-bits.lsp"
;; Res
;; Loa
;; 3 f                    al\\z.lsp"
;; Res
T
>
;; Loa
#P"c:\                    wlms.zal"
;; Loa
;; 18
#P"c:\                    wlms.zal"
;; Res                    OPERATION
T
> (ex)
*EX*
>
```

```
Lisp-Main-Window    Package: Zal    (a LISP-MAIN-WINDOW)
```

Figure 5-11 : Schema *NormalOperation*

The ZAL button opens an editing window to the executable version of the selected schema, and the TranZit button does the same for the original Z (Figure 5-12).


Allegro CL 3.0 [C:\ALLEGRO\allegro.img]
File Edit Search Window Packages Tools Builder Preferences Help
wlmspap.zed
wlmspap.zal   Package: zal
Start
Allegro CL 3.0 [C:\A...
01:50

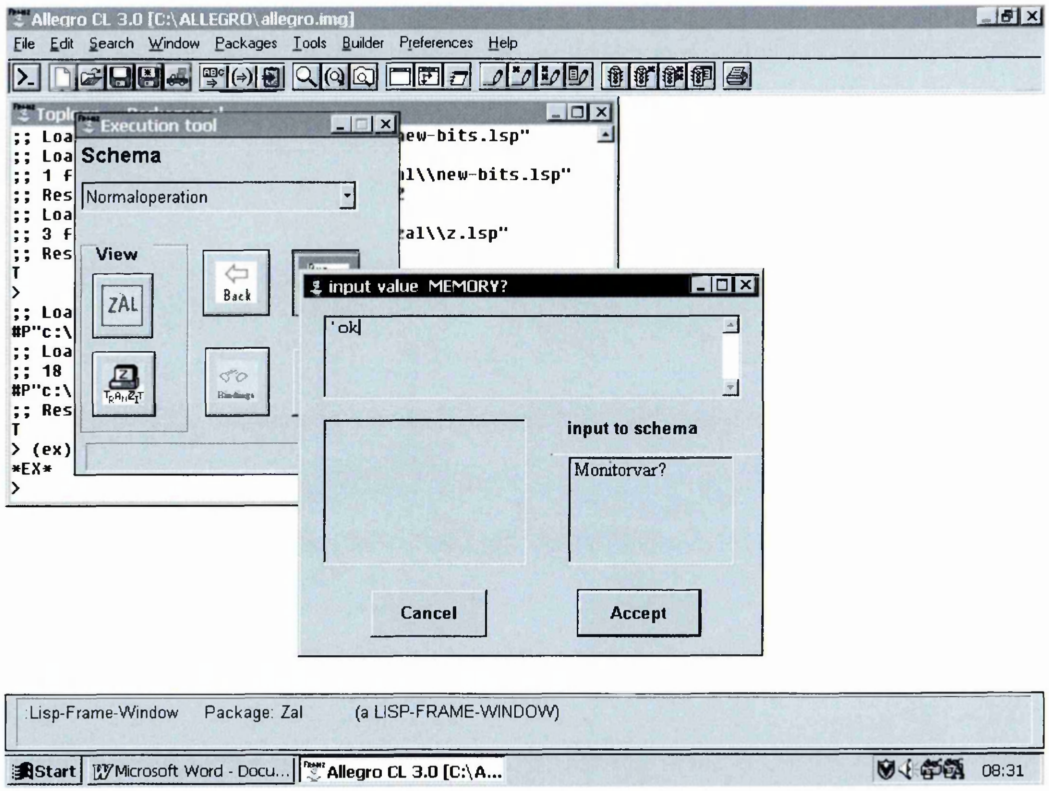
```
(SCHEMA NormalOperation
 :INCLUDE ( MonitorVar? delta_StoredData ControlSignals! GetNextMode AlarmCon
 :PREDICATE
 (and
      (eqz time' timeNow? )
      (let ((level (+ levelLowerCal
                      (floor (* (- (* diffPress? 103803 )485010 )
                                (- levelUpperCal levelLowerCal ))2550000 ))
           (eqz waterlevel
            (if (eqz diffPress? 0 )
               (- levelLowerCal 1 )
               (if (eqz diffPress? 255 )
                  (+ levelUpperCal 1 )
                  (if (\< level levelLowerCal )
                     levelLowerCal
                     (if (\> level levelUpperCal )
```

```
NormalOperation
MonitorVar?
ΔStoredData
ControlSignals!
GetNextMode
AlarmControl
GetOutputs

time' = timeNow?
(let level == levelLowerCal +
        ( (diffPress? * 103803 - 485010) * (levelUpperCal - levelLowerCal) ) div 2550000 •
        waterlevel =
                 if diffPress? = 0 then levelLowerCal - 1
```

```
Wize    Package: Zal    Path c:\mydocu1\research\papers\wlms\wlmspap.zed    228 lines and 8,223 characters    Last modified
Saturday 6 Nov 1999 at 1:37:04 PM (today)    (a TEXT-EDIT-WINDOW)
```

Figure 5-12 : Schema *NormalOperation* in Z and ZAL

Because NormalOperation makes reference (by schema inclusion) to the MonitorVar? schema, running it causes the ZAL system to prompt for values of the input variables (memory?, diffPress?, resetButton? etc.) defined by that schema (Figure 5-13). This use of schema inclusion demonstrates the flexibility of the approach to animation that has been adopted; support for the use of a schema to 'package' all the input variables in MonitorVar has not been explicitly provided. Given that the style of specification writing is consistent with the Z standard, it is implicitly supported by the toolset.



```
;; Loa                      ew-bits.lsp"
;; Loa
;; 1 f                      l\\new-bits.lsp"
;; Res
;; Loa
;; 3 f                      al\\z.lsp"
;; Res
T
>
;; Loa
#P"c:\
;; Loa
;; 18
#P"c:\
;; Res
T
> (ex)
*EX*
>
```

Schema: Normaloperation

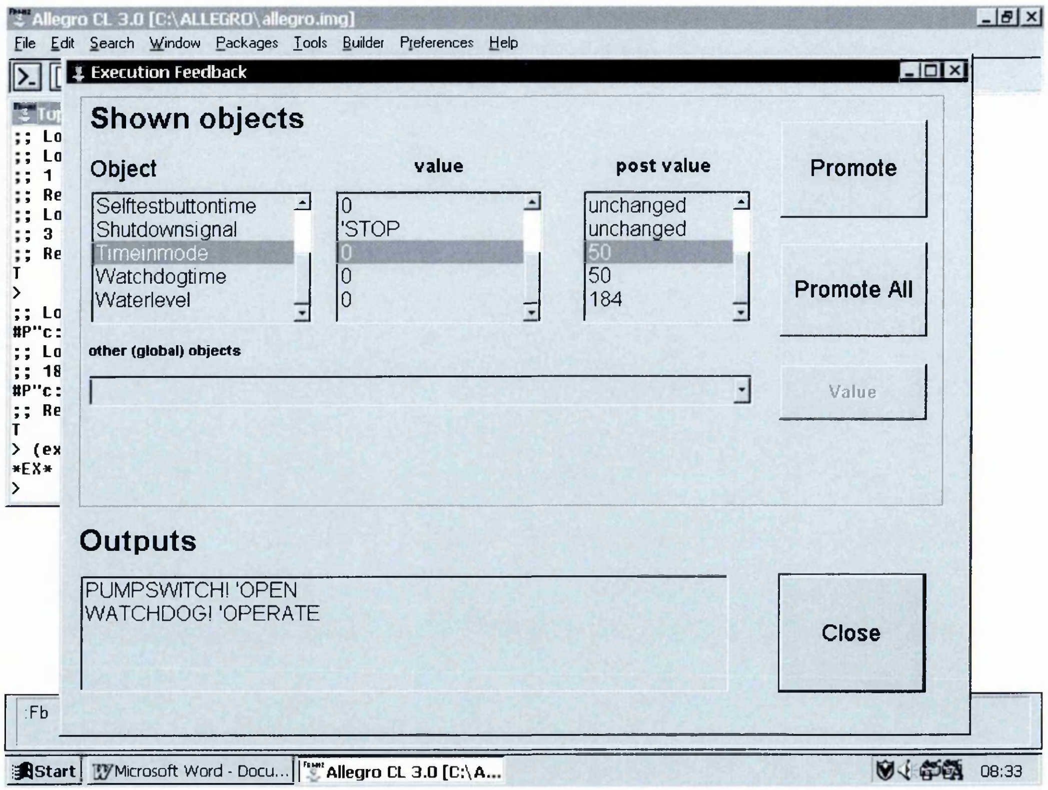
input value MEMORY?: 'ok

input to schema: Monitorvar?

Lisp-Frame-Window   Package: Zal   (a LISP-FRAME-WINDOW)

Figure 5-13 : Collecting input values to an included schema

After execution is complete, the before and after values of the system state variables are displayed, together with the values of the outputs pumpSwitch! and watchDog! (see).



Shown objects

Object	value	post value
Selftestbuttontime	0	unchanged
Shutdownsignal	'STOP	unchanged
Timeinmode	0	50
Watchdogtime	0	50
Waterlevel	0	184

other (global) objects

Outputs

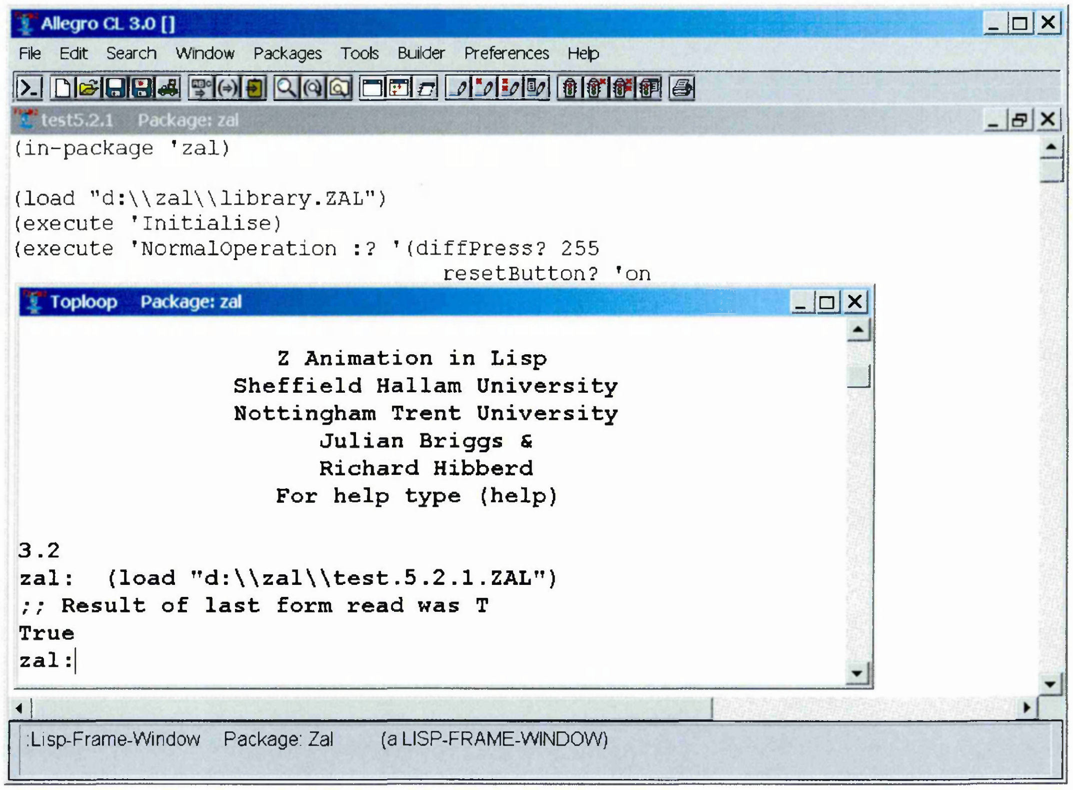
```
PUMPSWITCH! 'OPEN
WATCHDOG! 'OPERATE
```

Figure 5-14 : Execution outcomes displayed

This particular execution has also been effected through the command line interface, as was described in Section 2.4.2. The script file test5.2.1.ZAL is reproduced in Figure 5-15 and the outcome can be seen in Figure 5-16

```
    (in-package 'zal)
    (load "d:\\zal\\library.ZAL")
    (execute 'Initialise)
    (execute 'NormalOperation :? '(diffPress? 255
                                    resetButton? 'on
                                    selftestButton? 'no
                                    powerNow? 'on
                                    memory? 'ok
```

Figure 5-15 : ZAL Script File test5.2.1.ZAL

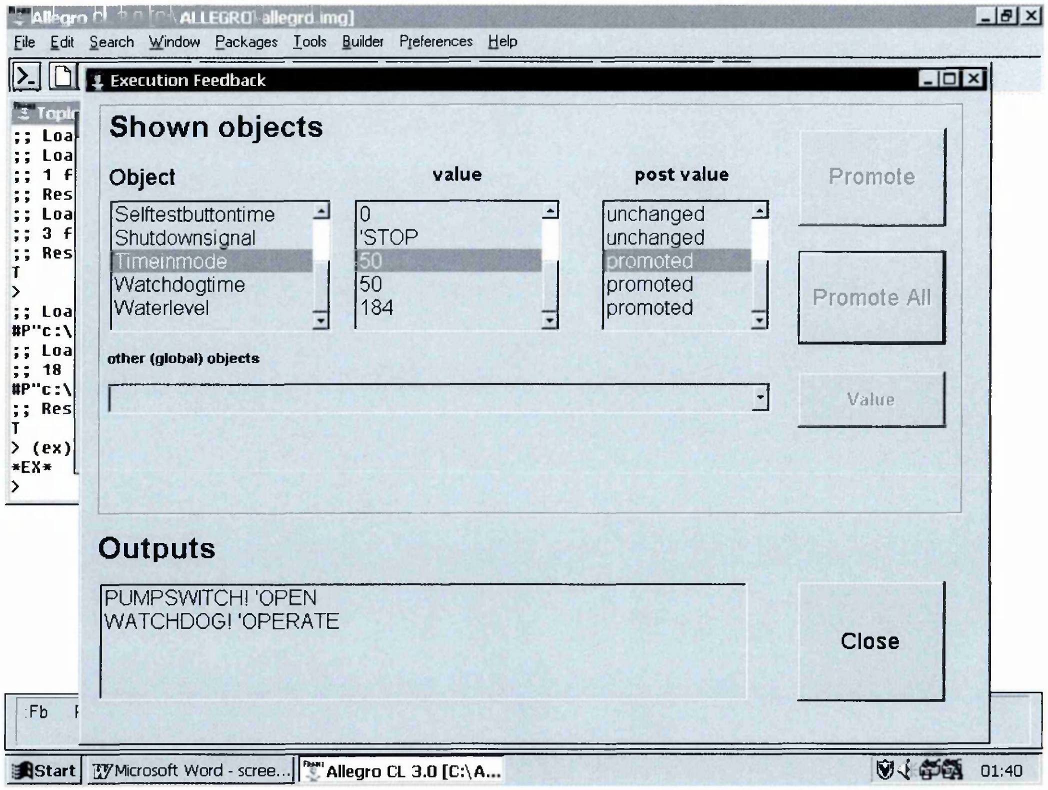

Allegro CL 3.0 []
File Edit Search Window Packages Tools Builder Preferences Help
test5.2.1 Package: zal
Toploop Package: zal
:Lisp-Frame-Window Package: Zal (a LISP-FRAME-WINDOW)

```
(in-package 'zal)

(load "d:\\zal\\library.ZAL")
(execute 'Initialise)
(execute 'NormalOperation :? '(diffPress? 255
                                resetButton? 'on
```

```
                     Z Animation in Lisp
                 Sheffield Hallam University
                 Nottingham Trent University
                       Julian Briggs &
                       Richard Hibberd
                    For help type (help)

3.2
zal:  (load "d:\\zal\\test.5.2.1.ZAL")
;; Result of last form read was T
True
zal:
```

Figure 5-16 : Automated execution using a script file

Clicking the Promote All button causes the system state variables to be promoted - that is, they take on their after state values (Figure 5-17).

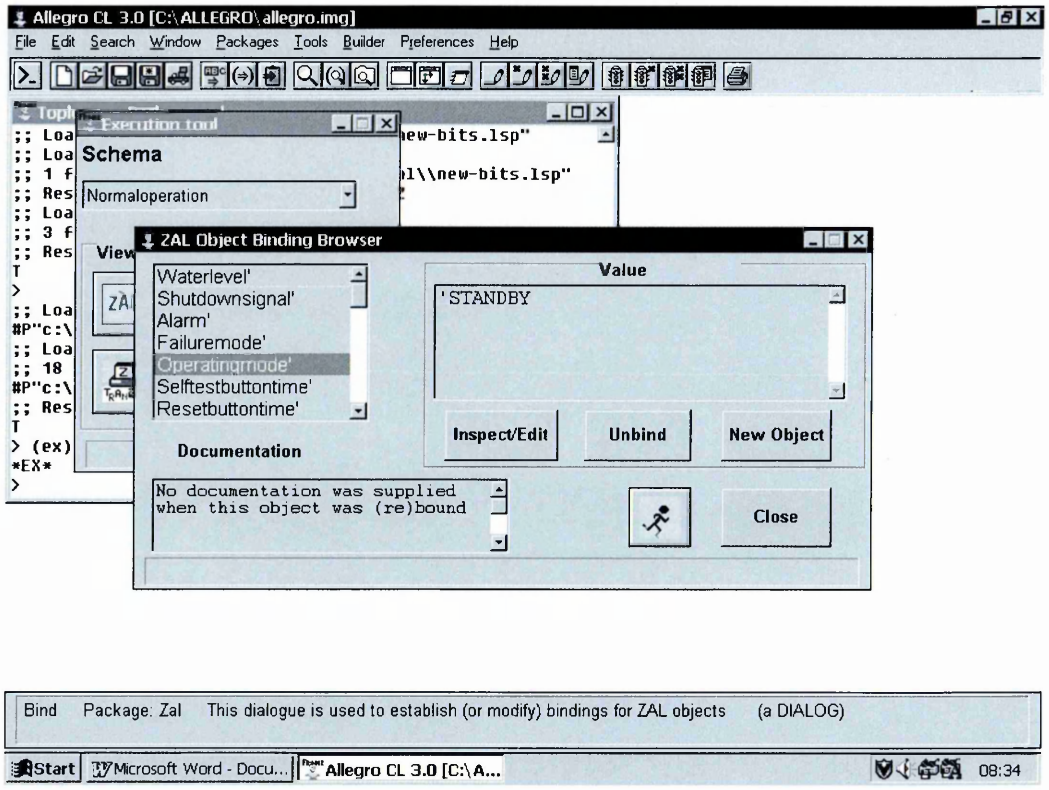

Allegro CL 3.0 [C:\ALLEGRO\allegrd.img]
File Edit Search Window Packages Tools Builder Preferences Help
Execution Feedback
Shown objects
Promote
Promote All
other (global) objects
Value
Outputs
Close
Start
Microsoft Word - scree...
Allegro CL 3.0 [C:\A...
01:40

Object	value	post value
Selftestbuttontime	0	unchanged
Shutdownsignal	'STOP	unchanged
Timeinmode	50	promoted
Watchdogtime	50	promoted
Waterlevel	184	promoted

```
PUMPSWITCH! 'OPEN
WATCHDOG! 'OPERATE
```

Figure 5-17 : Decorated values promoted

The Back button on the execution tool can be used to reverse this promotion if necessary. The Bindings button on the execution tool invokes the ``Binding Browser'' which can be used to inspect and edit current values of system state variables (Figure 5-18).



```
;; Loa
;; Loa
;; 1 f
;; Res
;; Loa
;; 3 f
;; Res
T
>
;; Loa
#P"c:\
;; Loa
;; 18
#P"c:\
;; Res
T
> (ex)
*EX*
>
```

```
new-bits.lsp"
l\\new-bits.lsp"
```

Waterlevel'
Shutdownsignal'
Alarm'
Failuremode'
Operatingmode'
Selftestbuttontime'
Resetbuttontime'

'STANDBY

No documentation was supplied when this object was (re)bound

Bind Package: Zal This dialogue is used to establish (or modify) bindings for ZAL objects (a DIALOG)

Figure 5-18 : The Binding Browser

Validating NormalOperation is an iterative procedure; each new execution prompts for new MonitorVar? inputs, and the system state variables and outputs are updated and displayed. The new system variable values are promoted by the user, and a new iteration begins. NormalOperation was validated against all of the test data described below. To speed this process, the input dialogue box keeps a history of the inputs previously used, thus allowing values to be selected rather than keyed in, if appropriate, as in (Figure 5-10).

The animation started in standby mode, and the inputs simulated the effect of the reset button being pressed for 3 seconds to start the pumps. During most of this time, the water level was within the normal limits, but for a short period in the middle it was allowed to become too low to check that this had no effect (exploratory investigation). The system went into operating mode and the reset button was released. The water level was then set too high and too low, in each case for less than 0.2 seconds, to check that the system entered shutdown mode but then returned to operating mode. During one of these periods, the reset button was pressed to check that it had no effect. The results of these tests are shown in more detail in Appendix D Table 1 Next the self test button was held down for 0.5 seconds, to send the system into test mode (see Appendix D Table 2). During this time the reset button was also held down for some time, and the water level was allowed to become too high for more that 0.2 seconds, to see what effect these conditions had. From test mode, the system should return to standby mode after 5 seconds. During part of this time the test data set the water level too low. From standby mode, the reset button was held down to return to operating mode, and then the water level became too high for more than 0.2 seconds to return the system to standby mode via shutdown mode. Next the test button was held down for long enough to send the system from standby to test mode and back again. The water level was too low for periods both in standby mode and test mode (see Appendix D Table 3). The reset button was then pressed to get the system to operating mode, and the water level was set too low for 0.4 seconds, but with the test button pressed for 0.5 seconds after less than 0.2 seconds of the low level, to see whether standby or test mode is entered (see Appendix D Table 4). This provocative investigation was carried out because of a suspicion, as a result of looking at the specification, that the system would not behave properly.

After a return to standby mode, the water pressure is set to 255, a value indicating that the water level monitoring device has failed. (This is also shown in Appendix D Table 4). The pressure is then allowed to return to a normal value, but the system should not return to normal operation. Then a gap of 0.5 seconds is left between inputs, which should result in a hardware failure being detected. Finally, the reset button is pressed for 3 seconds, with a normal water level, which should return the system to operating mode, but the previous detection of an irretrievable hardware error should keep the pumps switched off.

5.2.4 Discussion

Not surprisingly, the errors which were introduced as misprints in the translation from VDM to Z (see Section 5.2.1) were found immediately. The incorrect calculation of the water level resulted in levels which were too high by a factor of 10. The missing ``not" symbol resulted in a transition from operating mode to shutdown as soon as the former mode was entered with a water level within the correct limits. These errors were corrected to allow animation to continue so that more subtle errors or undesirable features might be exposed. Once the obvious errors had been corrected, the animation found no problems with the transition between operating modes. Where pressing buttons or allowing the water level to go outside its proper limits was expected to have no effect, this was found to be the case. However, possible problems were revealed with the values of the output variables controlling the pumps and the audible alarm.

Normally, when the water level becomes too high or too low in operating mode, shutdown mode is entered, but the pumps continue to operate for up to 0.2 seconds. If the water level returns to its proper limits within that time, the system returns to operating mode and the pumps are not switched off; if not, then standby mode is entered and the pumps are switched off. However, if the test button is held down while the system is in shutdown mode, it remains in that mode with the pumps on for 0.5 seconds before going to test mode.

The alarm should be audible when the water level is outside the proper limits and after standby mode is entered as a result of this, and it should be audible when there is a failure of the water level monitoring device, and for a period of 4 seconds when test mode is entered. The animation revealed that if the water level goes outside the limits and then recovers while the test button is held down in operating mode, the alarm does not become silent. More seriously, if the monitoring device fails during standby mode after test mode, the alarm remains silent.

If there is a failure of the water level monitoring device in test mode after the 4 seconds during which the alarm sounds, or in standby mode initially, then the alarm also stays silent. This particular problem was not found with the animation data used initially, though it was easily confirmed by a subsequent animation. These problems with output variables have safety implications. The parts of the specification concerned with these variables are unnecessarily complicated, and the problems are not obvious without a careful scrutiny of the specification.

The original paper [Williams 1994] that formed the motivation for this enquiry explored the relationship between the pump control and the pump environment. He found an ambiguity in the Software Cost Reduction (SCR) specification [van Schouwen 1991] because it is incomplete, in the sense that the behaviour of one of the environment constraining variables is not specified; hence the safety of the WLMS could be affected because the pumps cannot be properly shut down. In contrast this exercise focuses on the transitions between the different operating modes; once the WLMS has been initialised it is in exactly one of the following modes: operating, standby, shutdown or test. This investigation reveals a number of errors in the specification as it stands, some of which can lead to safety being compromised because the alarm remains silent despite the occurrence of failures. Jackson and Stokes' work demonstrated the effectiveness of using formal specification to model requirements and hence develop confidence in our understanding of the safety-critical system specified; building on that work, this exercise demonstrates the effectiveness of the REALiZE method in general and of the tool components in furthering our understanding of the water level monitoring system. In particular, it was possible to detect a number of errors that had remained undetected in Jackson and Stokes' formalisation. Given the unnecessarily complicated specification structure, the errors were not obvious, and their consequences might not be revealed even with a careful scrutiny of the specification. However, exercising the specification through animation allowed the confirmation of the intended consequences and the identification of the unforeseen ones.

5.3 A Car Rental System

The problem examined here derives from a coursework assignment required of Final Year undergraduates, studying Software Engineering at Sheffield Hallam University. The brief (Appendix B (Car Rental Specification) is intentionally non-prescriptive so as not to constrain or unduly direct the efforts of the students, and offers scope for a wide range of models without strongly suggesting any particular one. As this will usually be the first specification of any size attempted by the students, some of the resulting solutions are not the most obvious that might occur to more experienced specifiers; it is the case though that the toolset can support development and animation of a great variety of quite complex Z. One such specification is considered here, together with a description of how the animator can be used to refine it into an alternative form; for a more detailed description of the student experience of using the REALiZE toolset, the reader is referred [Siddiqi *et al* 1998]. The problem description is presented, together with a solution in both Z and ZAL. Some observations are made on this solution and then some improvements are suggested; in each case the "improved" version is suffixed '2'. The process of deriving these improvements is also examined, along with the contribution that the animator can make to this process. As the focus here is predominately on the ZAL code, the following description will concentrate on this and its development rather than its execution.

The following describes the keys aspects of the required specification

> *"The ACME car hire company has a fleet of cars distributed across the country in a number of depots in several major cities. Each car is identified by a unique id number. The depot in which the car is currently garaged, the car's current mileage and the car's manufacturer is recorded for each car. A client can hire a car from any depot and return it to any other depot. When a customer hires a car from a particular depot, they provide their name and specify the make of car they want, and the hire date is recorded. A specific car (if available) is then allocated to them. When the car is returned (possibly to another depot), its new mileage is inspected, and the customer is charged 10p per mile plus a fixed charge of £20 for each day of the hire period. The ACME company is so successful that sometimes there's a queue of people waiting to return their hired car to a depot. When this happens, the company deals with the queue in strict order. Occasionally a customer in the queue gets tired of waiting and leaves, hoping to return later when the queue is shorter.*

The student task is to develop a specification for this system, to include a description of hiring and returning a car, and to allow a number of specific queries regarding the location and availability of vehicles (see Appendix B (Car Rental Specification).

The specification discussed here is a student submission, which is developed in a number of ways to illustrate features of the ZAL environment.

CarRental is the state schema modelling this system

CarRental

$$
\begin{array}{l}
Cars : CARID \nrightarrow MANUFACTURER \\
Depots : \mathbb{P}\ DEPOT \\
CarsInDepot : CARID \nrightarrow DEPOT \\
CarMileage : CARID \nrightarrow \mathbb{N} \\
Hired : CARID \nrightarrow NAME \\
HiredDate : CARID \nrightarrow \mathbb{N} \\
Returning : \mathrm{seq}\ CARID \\
ReturningDepot : CARID \nrightarrow DEPOT \\
\hline
\mathrm{dom}\ Cars = \mathrm{dom}\ CarMileage \\
\forall\ c{:}CARID \mid c \in \mathrm{dom}\ Hired \bullet c \notin \mathrm{dom}\ CarsInDepot \\
\mathrm{dom}\ Hired = \mathrm{dom}\ HiredDate \\
\mathrm{dom}\ Hired \cup \mathrm{dom}\ CarsInDepot = \mathrm{dom}\ Cars \\
\mathrm{ran}\ CarsInDepot \subseteq Depots \\
\mathrm{ran}\ Returning \subseteq \mathrm{dom}\ Hired \\
\mathrm{ran}\ ReturningDepot \subseteq Depots \\
\mathrm{dom}\ ReturningDepot = \mathrm{ran}\ Returning \\
\#Returning = \#ReturningDepot \\
\ \forall\ i,j : \mathbb{N} \mid i \in \mathrm{dom}\ Returning \land j \in \mathrm{dom}\ Returning \\
\qquad\qquad \bullet\ i \neq j \Rightarrow Returning(i) \neq Returning(j) \\
\forall\ c : CARID \mid c \in \mathrm{dom}\ Cars \bullet \#(\{c\} \lhd Cars) = 1 \\
\forall\ c : CARID \mid c \in \mathrm{dom}\ CarMileage \bullet \#(\{c\} \lhd CarMileage) = 1 \\
\forall\ c : CARID \mid c \in \mathrm{dom}\ Hired \bullet \#(\{c\} \lhd Hired) = 1 \\
\forall\ c : CARID \mid c \in \mathrm{dom}\ HiredDate \bullet \#(\{c\} \lhd HiredDate) = 1 \\
\forall\ c : CARID \mid c \in \mathrm{dom}\ CarsInDepot \bullet \#(\{c\} \lhd CarsInDepot) = 1
\end{array}
$$

The final five predicates are all quantifications designed to ensure that, respectively, Cars CarMileage Hired HiredDate and CarsInDepot are functions; this is already a requirement as each of them is declared as a function. However this type information is not used by the animator and the specifier has chosen the quantifications as a valid if complicated way of expressing this. Two alternatives suggest themselves: the first is to use the same approach, but with a simpler predicate such as $\#Cars = \#(\mathrm{dom}\ Cars)$;

the second is to rectify the omission where it is omitted i.e. in the ZAL code, which requires a simple call to the ZAL predicate `mapp` which tests that its argument is indeed a function. The second approach will require the same edit of the ZAL code every time the Transformation Engine is used to update / create ZAL code, but this can be readily accomplished by abstracting all such constraints to a single schema which can be used in much the same way as an abstraction schema is utilised in data refinement. Though no part of the work reported here, it is considered it would be a relatively simple matter to generate this type constraining code using the range of ZAL predicates `seqp, mapp, relp, setp, injectivep` which test their argument is a sequence, function, relation, set and injective function respectively.

The decision to maintain a single queue of returning vehicles for all depots leads to the over-complication of the Return operation and also Query3; notwithstanding this, these schemas can be readily animated, as will be seen. The ZAL code for the state schema is

```
(SCHEMA CarRental
 :PREDICATE
 (and
      (eqz (dom Cars )(dom CarMileage ))
      (forall   c (dom CarMileage )
       (imply (mem c (dom CarMileage )) (eqz (card (domres  {c  }CarMileage ))1 )))
      (forall c (dom Cars)
         (imply (mem c (dom Cars ))(eqz (card (domres  {c  }Cars )) 1 )))
      (forall   c (dom Hired )
         (imply (mem c (dom Hired )) (not-mem c (dom CarsInDepot ))))
      (eqz (dom Hired )(dom HiredDate ))
      (forall   c (dom Hired )
         (imply  (mem c (dom Hired )) (eqz (card (domres  {c  }Hired )) 1 ) ) )
      (forall   c (dom HiredDate )
         (imply (mem c (dom HiredDate )) (eqz (card (domres  {c  }HiredDate ))1 )))
      (eqz (unionz (dom Hired )(dom CarsInDepot ))(dom Cars ))
      (subset (ran CarsInDepot )Depots )
      (forall   c (dom CarsInDepot )
         (imply (mem c (dom CarsInDepot )) (eqz (card (domres  {c }CarsInDepot ))1 )))
      (subset (ran Returning )(dom Hired ))
      (subset (ran ReturningDepot )Depots )
      (eqz (dom ReturningDepot )(ran Returning ))
      (eqz (card Returning )(card ReturningDepot ))
      (and ;; not in Z
           (forall  j (dom Returning )  i (dom Returning )
            (imply (neqz i j ) (neqz (applyz Returning i )(applyz Returning j ))
             ) ) ) ) )
```

It can be seen that roughly half of the predicates are concerned solely with modeling the type constraints of the functions and this will obviously militate adversely the readability of the specification.

The possible improvements to this schema are largely those suggested earlier, utilising `mapp`, and to use `disjoint` rather than a quantification to require a vehicle to be either hired or in the depot.

```
(SCHEMA CarRental2
 :PREDICATE
 (and
      (eqz (dom Cars )(dom CarMileage ))
      (disjoint < (dom Hired) (dom CarsInDepot) > )
      (eqz (dom Hired )(dom HiredDate ))
      (eqz (unionz (dom Hired )(dom CarsInDepot ))(dom Cars ))
      (subset (ran CarsInDepot )Depots )
      (subset (ran Returning )(dom Hired ))
      (subset (ran ReturningDepot )Depots )
      (eqz (dom ReturningDepot )(ran Returning ))
      (eqz (card Returning )(card ReturningDepot ))
      (eqz (card Returning) (card (ran Returning)))
      ; the following six predicates constrain their arguments to be functions
      (mapp CarsInDepot)
      (mapp Hired)
      (mapp HiredDate)
      (mapp CarMileage)
      (mapp Cars)
      (mapp Hired)
      )
)
```

The operation schema Hire describes the behaviour expected of a successful hiring and might be called HireOK if a total operation with error reporting were required; the operation

- checks the inputs are valid values
- confirms the availability of the required make at the particular depot
- updates

  Hired

  HiredDate

  CarsInDepot

- constrains to not change

  Cars

  CarMileage

  Depots

  Returning

  ReturningDepot

This gives

```
┌Hire──────────────────────────────────────
│ΔCarRental
│name? : NAME
│man? : MANUFACTURER
│depot? : DEPOT
│date? : ℕ
├──────────────────────────────────
│depot? ∈ Depots
│man? ∈ ran Cars
│(∃ c : CARID | c ∈ dom Cars
│   • man? = Cars(c) ∧ c ∈ dom CarsInDepot ∧ depot? = CarsInDepot(c))
│Hired' = Hired ∪ {c[11] ↦ name?}
│HiredDate' = HiredDate ∪ {c ↦ date?}
│CarsInDepot' = {c} ⩤ CarsInDepot)
│Cars' = Cars
│CarMileage' = CarMileage
│Depots' = Depots
│Returning' = Returning
│ReturningDepot' = ReturningDepot
└──────────────────────────────────────────
```

The reader can observe that this description is flawed, since the object c which is used to record the actual vehicle hired is not in scope when it is used; this could be rectified by conjoining the updating predicates with those in the quantification or else by declaring a local variable of type CARID, either of which approach could correct Z which would translate correctly. The presented usage, together that of the Lisp primitive `setf` to capture a value that satisfies the quantification, indicates that the ZAL version was developed interactively, and then reverse-engineered to generate the Z equivalent.

[11] c is not in scope here.

```
(SCHEMA Hire
 :? ( name? man? depot? date?)
 :INCLUDE delta_CarRental
 :SHOW (HiredDate Hired CarsInDepot )
 :PREDICATE
 (and
      (mem depot? Depots )
      (mem man? (ran Cars ))
      (exist   c (dom Cars )
       (and
            (mem c (dom Cars ))
            (and
                 (equalp man? (applyz Cars c ))
                 (mem c (dom CarsInDepot))
                 (equalp depot? (applyz CarsInDepot c))
                 (setf Hired' (unionz Hired  { #(c name? ) }))
                 (setf HiredDate' (unionz HiredDate  { #(c date? ) }))
                 (setf CarsInDepot' (domsub  {c  }CarsInDepot ))
                 )
            )
       )
      (eqz Cars' Cars )
      (eqz CarMileage' CarMileage )
      (eqz Depots' Depots )
      (eqz Returning' Returning )
      (eqz ReturningDepot' ReturningDepot )
      )
 )
```

Notwithstanding these corrections, the Hire operation remains unwieldy and not particularly natural; this follows directly from the original choice of state schema.

The modified second version dispenses with the quantification at the root of the original difficulty; the availability is determined by $(man? \mapsto depot?) \in (Cars^{\sim} \fatsemi CarsInDepot)$,

i.e. constructing the MANUFACTURER × DEPOT tuples by relational composition. An

actual vehicle is chosen using the ZAL function `pick`, which selects a randomly-chosen element from a set; in reality, there would almost certainly be some alternative mechanism for selecting which one of a number of possible vehicles is selected. As was discussed in Section **4.1**, `pick` is implemented with the inbuilt Lisp function `random`, but a non-deterministic version is also available. While `pick` has no direct equivalent in Z, the `:LOCALS` keyword is used to introduce a temporary "variable" and would normally be generated from a Z let expression; the set of candidate vehicles from which one is selected to be hired is described by

$\mathrm{dom}\ (\mathrm{Cars} \rhd \{\mathrm{man?}\}) \cap \mathrm{dom}\ (\{\mathrm{depot?}\} \lhd \mathrm{CarsInDepot})$ i.e. those with both the correct manufacturer and at the depot in question.

```
(SCHEMA Hire2
    :? ( name? man? depot? date?)
    :INCLUDE delta_CarRental
    :SHOW (HiredDate Hired CarsInDepot )
    :LOCALS c
    :PREDICATE
    (and
        (mem depot? Depots )
        (mem man? (ran Cars ))
        (mem #(man? depot?) (rel-compose (inverse Cars) CarsInDepot))
        (eqz c
            (pick (inter (dom (ranres cars {man?} ))
                  (dom (ranres carsInDepot {depot?} )))))
        (eqz Hired' (unionz Hired  { #(c name? ) }))
        (eqz HiredDate' (unionz HiredDate  { #(c date? ) }))
        (eqz Cars' Cars )
        (eqz CarMileage' CarMileage )
        (eqz Depots' Depots )
        (eqz Returning' Returning )
        (eqz ReturningDepot' ReturningDepot )
        )
    )
```

Returning a car to a depot is achieved in this model by two operations: the first is to join a queue of returners at some depot and the second is to process the first returning vehicle queuing at a particular depot; as a consequence of using a single sequence for all returning vehicles, updating requires a squash of this sequence rather than just taking the tail of it. A further complication is that the queue at any one depot must be extracted using range restriction as in $(\mathrm{Returning} \subset (\mathrm{dom}\ (\mathrm{ReturningDepot} \rhd \{\mathrm{depot?}\}\ ))$.

The operations are described thus

AddReturningCar

ΔCarRental

depot? : DEPOT

car? : CARID

---

depot? $\in$ Depots

car? $\in$ dom Hired

car? $\notin$ ran Returning

Returning' = Returning $\frown$ $\langle$car?$\rangle$

ReturningDepot' = ReturningDepot $\cup$ {car? $\mapsto$ depot?}

Cars' = Cars

CarMileage' = CarMileage

CarsInDepot' = CarsInDepot

Hired' = Hired

HiredDate' = HiredDate

Depots' = Depots

This translation of this operation by TranZit is entirely straightforward – the inputs are validated and the details are added to Returning and ReturningDepot; there is no "improved" version of this operation.

```
(SCHEMA AddReturningCar
    :? ( depot? car?)
    :INCLUDE delta_CarRental
    :SHOW (ReturningDepot Returning )
    :PREDICATE
    (and
            (mem depot? Depots )
            (mem car? (dom Hired ))
            (not-mem car? (ran Returning ))
            (eqz Returning' (appendz Returning  <car?  >))
            (eqz ReturningDepot' (unionz ReturningDepot  { #(car? depot? ) }))
            (eqz Cars' Cars )
            (eqz CarMileage' CarMileage )
            (eqz CarsInDepot' CarsInDepot )
            (eqz Hired' Hired )
            (eqz HiredDate' HiredDate )
            (eqz Depots' Depots )
            )
    )
```

The complexity of the processing of returning a vehicle under this model is found in the operation to accept the first vehicle at a given depot back into "stock"; after the inputs are validated, almost all the data components must be updated:

- the actual vehicle is identified as the head of the queue at the depot in question;
- the new mileage is recorded;
- the cost is calculated from the difference in mileage and in dates;
- the vehicle must be removed from `Returning` and `ReturningDepot` and also from `Hired` and `HiredDate`;
- and the vehicle is added to `CarsInDepot`.

```
┌ReturnByDepot─────────────────────────────
│ΔCarRental
│depot? : DEPOT
│date? : ℕ
│mileage? : ℕ
│bill! : ℕ
│car: CARID
├──────────────────────────────
│depot? ∈ Depots
│depot? ∈ ran ReturningDepot
│Returning ≠ ⟨⟩
│car = (squash (Returning ▷ (dom (ReturningDepot ▷ {depot?} )))) 1
│date? ≥ HiredDate(car)
│mileage? ≥ CarMileage(car)
│bill! = ((date? - HiredDate(car)) * 20) + 1 * (mileage? -CarMileage(car))
│Returning' = squash(Returning ⩥ {car})
│ReturningDepot' = {car} ⩤ ReturningDepot
│Hired' = {car} ⩤ Hired
│HiredDate' = {car} ⩤ HiredDate
│CarsInDepot' = CarsInDepot ∪ {car ↦ depot?}
│CarMileage' = CarMileage ⊕ {car ↦ mileage?}
│Cars' = Cars
│Depots' = Depots
└──────────────────────────────
```

Again this is a straightforward transformation and there is no improved version, though note the inconsistency in the way that the returning vehicle is extracted from the squashed Returning function, by a function application in the Z version, and by using head in the ZAL version. This would again indicate that ZAL code has been executed and edited, rather than generated from the Z at each stage.

```
(SCHEMA ReturnByDepot
 :? ( depot? date? mileage?)
 :! bill!
 :INCLUDE delta_CarRental
 :SHOW (ReturningDepot Returning HiredDate Hired CarsInDepot CarMileage )
 :PREDICATE
 (and
      (mem depot? Depots )
      (mem depot? (ran ReturningDepot ))
      (neqz ReturningDepot <>)
      (eqz car
           (head (squash (ranres Returning (dom (ranres ReturningDepot  {depot?  })))))))
      (\>= date? (applyz HiredDate car ))
      (\>= mileage? (applyz CarMileage car ))
      (eqz bill!
           (+  (* (- date? (applyz HiredDate car ))2000 )
               (* 10 (- mileage? (applyz CarMileage car )))))
      (eqz Returning' (squash (ransub Returning  {car  })))
      (eqz ReturningDepot' (domsub  {car  }ReturningDepot ))
      (eqz Hired' (domsub  {car  }Hired ))
      (eqz HiredDate' (domsub  {car  }HiredDate ))
      (eqz CarsInDepot' (unionz CarsInDepot  { #(car depot? ) }))
      (eqz CarMileage' (override CarMileage  { #(car mileage? ) }))
      (eqz Cars' Cars )
      (eqz Depots' Depots )
      )
 )
```

None of the query operations alters the state and consequently their specifications include ΞCarRental; this schema expands to

```
(SCHEMA psi_CarRental
    :INCLUDE (CarRental CarRental' )
    :PREDICATE
    (and
            ( eqz ReturningDepot ReturningDepot' )
            ( eqz Returning Returning' )
            ( eqz HiredDate HiredDate' )
            ( eqz Hired Hired' )
            ( eqz Depots Depots' )
            ( eqz CarsInDepot CarsInDepot' )
            ( eqz Cars Cars' )
            ( eqz CarMileage CarMileage' )
            )
    )
```

Query1 has to join Cars and CarsInDepot to generate the appropriate set of Depot; this can be done using domain and range restriction as below:

```
┌Query1──────────────────────────────────
│ΞCarRental
│man? : MANUFACTURER
│result! : ℙ DEPOT
├─────────────────────────────
│man? ∈ ran Cars
│result! = ran (dom (Cars ▷ {man?}) ◁ CarsInDepot)
└────────────────────────────────────────
```

The expression $\mathrm{ran}\ (\mathrm{dom}\ (Cars \rhd \{man?\}) \lhd CarsInDepot)$

can be derived by reasoning and then confirmed correct by animation, but an alternative technique is available to the user who is familiar with ZAL. It is often useful to interact with the animator at a command line where native ZAL expressions can be entered and evaluated. Once the values of data objects have been instantiated, the expressions in question can be derived incrementally; these will of course require confirmation, but in a situation such as Query1 posits, it is useful to experiment with possible expressions. One such dialogue is presented now, though with only correct expressions.

```
zal: Cars
{('C1 'PONTIAC) ('C2 'MINI) ('C3 'SKODA) ('C4 'SKODA) ('C5 'FIAT) ('C6 'SKODA)}
zal: CarsInDepot
{('C1 'DERBY) ('C2 'DERBY) ('C4 'SHEFFIELD) ('C6 'DERBY)}
zal: (ranres Cars {'SKODA})
[#('C3 'SKODA) #('C4 'SKODA) #('C6 'SKODA)]
zal: (dom (ranres Cars {'SKODA}))
{'C3 'C4 'C6}
zal: (domres  (dom (ranres Cars {'SKODA})) CarsInDepot)
[#('C4 'SHEFFIELD) #('C6 'DERBY)]
zal: (ran  (domres   (dom (ranres Cars {'SKODA})) CarsInDepot))
{'DERBY 'SHEFFIELD}
zal: (ran  (domres   (dom (ranres Cars {'MINI})) CarsInDepot))
{'DERBY}
zal: (ran  (domres   (dom (ranres Cars {'PONTIAC})) CarsInDepot))
{'DERBY}
zal: (ran  (domres   (dom (ranres Cars {'FIAT})) CarsInDepot))
{}
zal:
```

This dialogue is more useful if the Lisp facility to use the result of the previous evaluation, accessed as *, is utilised; the sequence of expressions is also clearer:

```
zal: (ranres Cars {'SKODA})
[#('C3 'SKODA) #('C4 'SKODA) #('C6 'SKODA)]
zal: (dom *)
{'C3 'C4 'C6}
zal: (domres * CarsInDepot)
[#('C4 'SHEFFIELD) #('C6 'DERBY)]
zal: (ran *)
{'DERBY 'SHEFFIELD}
zal:
```

In fact this approach facilitates the development of an equivalent expression using relational image

```
zal: (inverse cars)
[#('FIAT 'C5) #('MINI 'C2) #('PONTIAC 'C1) #('SKODA 'C3) #('SKODA 'C4) #('SKODA 'C6)]
zal: (rel-image {'SKODA} *)
{'C3 'C4 'C6}
zal: (rel-image * CarsInDepot)
{'DERBY 'SHEFFIELD}
zal:
```

This leads to the expression

$$\text{result!} = (\text{Cars}^{\sim} ⦇ \{\text{man?}\} ⦈) ⦇ \text{CarsInDepot} ⦈$$

and thence to

```
(SCHEMA Query1
:? man?
:! result!
:INCLUDE psi_CarRental
:SHOW (cars carsindepot)
:PREDICATE
(and
(mem man? (ran Cars ))
;; use either one of the following expressions
(eqz result! (ran (domres (dom (ranres Cars  {man?  }))CarsInDepot )))
;; (eqz result! (rel-image (rel-image {man?} (inverse Cars) )CarsInDepot))
)
)
```

In a similar way, the description of Query2 can be developed interactively; the initial version uses a set comprehension to describe the vehicles currently at a given depot, and in many ways this is a natural description – those vehicles that satisfy the predicate *are at depot d.*

```
┌Query2──────────────────────────────────────
│ΞCarRental
│depot? : DEPOT
│result! : ℙ CARID
├─────────────────────────────
│depot? ∈ Depots
│result! = {c:CARID | c ∈ dom CarsInDepot ∧ CarsInDepot(c) = depot?}
└────────────────────────────────────
```

This gives a ZAL version

```
(SCHEMA Query2
 :? depot?
 :! result!
 :INCLUDE psi_CarRental
 :PREDICATE
 (and
      (mem depot? Depots )
      (eqz result!
       (mksi 'c  'c (dom Cars)
        '(and
              (mem c (dom CarsInDepot ))
              (equalp (applyz CarsInDepot c )depot? )

              )
        )
       )
      )
 )
```

As an alternative either of

result! = dom (CarsInDepot $\rhd$ {depot?} and

result! = CarsInDepot$^{\sim}$ $(\!|$ {depot?} $|\!)$

will give the same set of vehicles as in

```
(SCHEMA Query2.2
 :? depot?
 :! result!
 :INCLUDE psi_CarRental
 :PREDICATE
 (and
      (mem depot? Depots )
      ;;either
      (eqz result! (dom (ranres CarsInDepot {depot?})))
      ;; or
      (eqz result! (rel-image  {depot?} (inverse CarsInDepot ) ) )
      )
 )
```

Again the interaction to derive the expressions is illustrative:

```
zal: (ranres CarsInDepot {'DERBY})
[#('C1 'DERBY) #('C2 'DERBY) #('C6 'DERBY)]
zal: (dom *)
{'C1 'C2 'C6}
zal:
```

and

```
zal: (rel-image {'DERBY} (inverse CarsInDepot))
{'C1 'C2 'C6}
zal:
```

Query3 again involves filtering the sequence Returning to establish a queue for a single depot; this queue must be composed in some way with function Cars which maps vehicles to manufacturers. The actual query can then be readily answered by using a relational image, given the types

Cars : CARID $\rightarrowtail$ MANUFACTURER

Returning : seq CARID , though this again needs to be restricted to a particular depot and then squashed.

The relation of interest is from MANUFACTURER to ℕ; this will give the tuples of

manufacturers and the position in the queue at this depot. This can be achieved either by inverting the composition or by composing the inverse of both or else by using backward composition i.e.

```
┌Query3──────────────────────────────────────
│ΞCarRental
│depot? : DEPOT
│man? : MANUFACTURER
│place! : ℙ ℕ
│X : seq CARID
├──────────────────────────────────
│depot? ∈ ran ReturningDepot
│man? ∈ ran Cars
│Returning ≠ ⟨⟩
│X = squash (Returning ▷ (dom (ReturningDepot ▷ {depot?})))
│/* either */
│place! = {man?} ⦇ (X ⨾ cars)˜ ⦈
│/* or place! = {man?} ⦇ (cars˜⨾ X˜) ⦈ */
│/* or backward composition place! = {man?} ⦇ (cars ∘ X) ˜ ⦈ */
└──────────────────────────────────────────
```

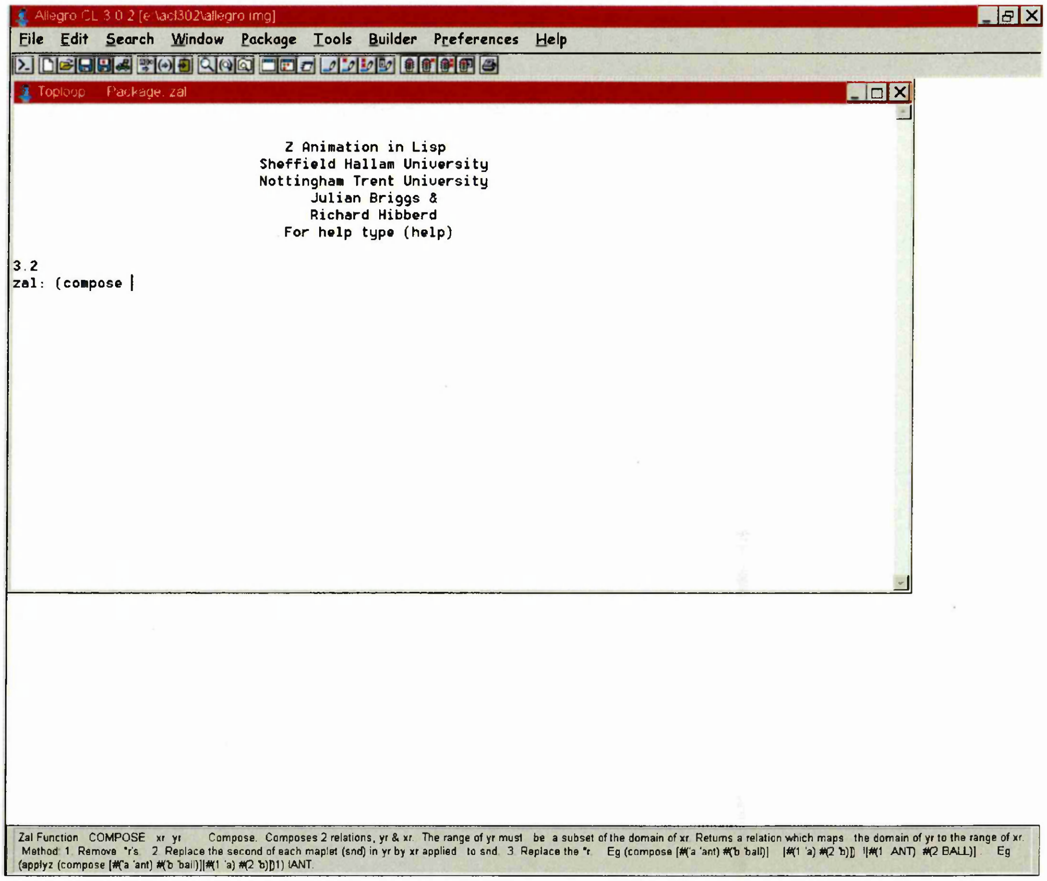
This gives the following ZAL code

```
(SCHEMA Query3
 :? ( depot? man?)
 :! place!
 :INCLUDE psi_CarRental
 :LOCALS (temp)
 :PREDICATE
 (and
      (mem depot? (ran ReturningDepot ))
      (mem man? (ran Cars ))
      (eqz temp
           (squash (ranres Returning (dom (ranres ReturningDepot  {depot?  })))))
      (eqz place!
          (inverse (rel-compose temp Cars ))
       ;;;           ;; alternative model
       ;;;              (rel-compose
       ;;;                (inverse Cars)
       ;;;                (inverse temp)))

       ;;;           ;; 2nd alternative backward composition
       ;;;            (rel-image {'fiat} (inverse (compose cars temp)))
       )
      )
 )
```

Of interest again is the command line manipulation

```
zal:  ;the following generates the returning queue at Sheffield
zal: (squash (ranres Returning (dom (ranres ReturningDepot  {'sheffield  }))))
<'C3 'C5>
zal: ;this is composed with Cars to get the Sheffield queue as sequence of Manufacturer
zal: (compose Cars *)
[#(1 'SKODA) #(2 'FIAT)]
zal: ; take the relational image of the set {'FIAT}
zal: (rel-image {'FIAT} *)
Zal error: the set, xs, passed to domres must be
of the same type as the domain of xm.
xs is {'FIAT}
xm is [#(1 'SKODA) #(2 'FIAT)]
zal: ; the error indicates the composition gives ℕ ↔ MANUFACTURER, which should

have been inverted to give MANUFACTURER ↔ ℕ; the * refers to last result and

the ** to the last-but-one result
zal:  (rel-image {'fiat} (inverse **))
{2}
zal:  (rel-image {'fiat} (inverse (compose cars (squash (ranres Returning (dom (ranres
ReturningDepot  {'sheffield  })))))))
{2}
zal:
```

The command line use of the execution engine is helped greatly by the Lisp environment in which it is hosted; the status bar of the application window provides guidance to the user by displaying relevant information as the user types. The following screens show two stages the entry of an expression; in the first can be seen the parameter pattern (or lambda list) expected by the function compose, and also the first two lines of the documentation string.

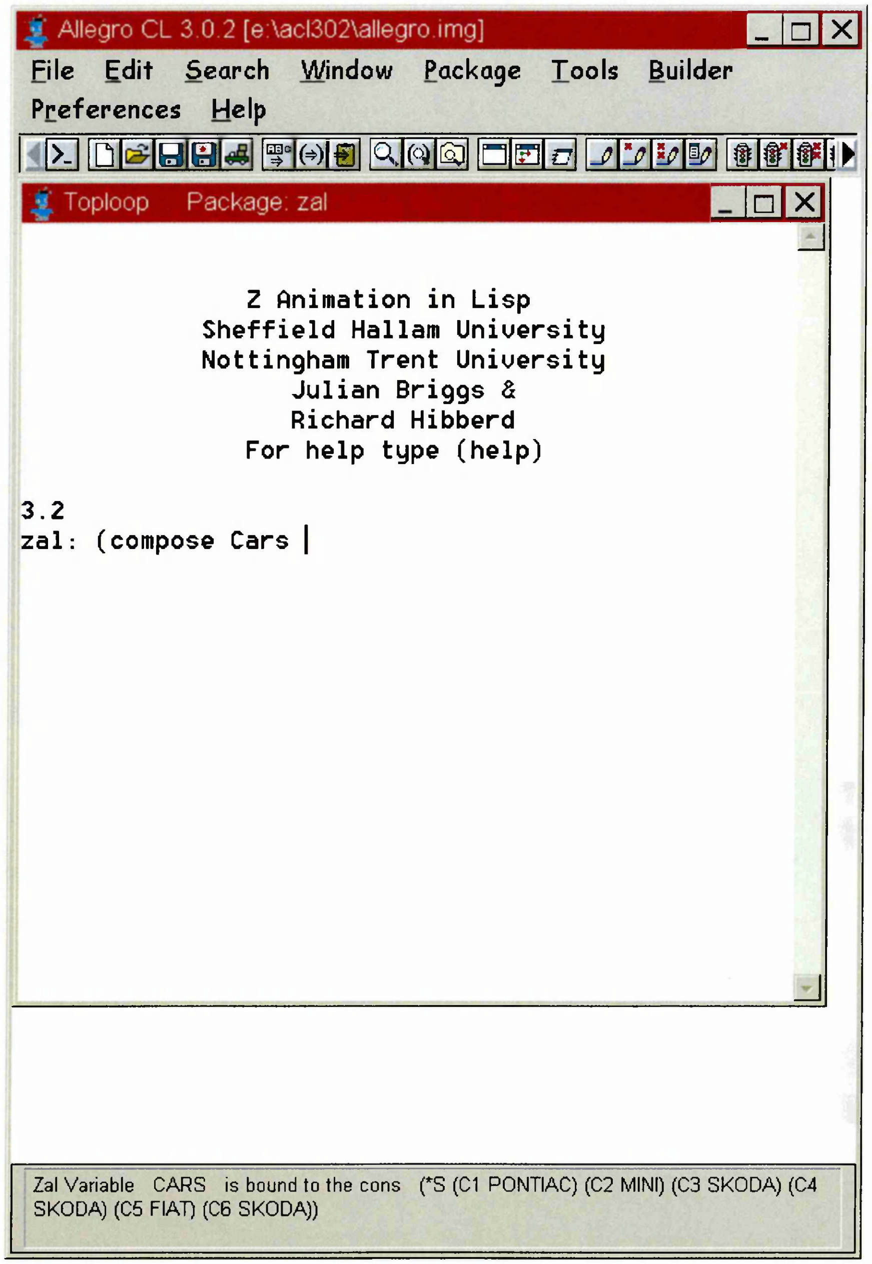

Allegro CL 3.0.2 [e:\acl302\allegro.img]
File Edit Search Window Package Tools Builder Preferences Help
Toploop Package: zal

```
Z Animation in Lisp
Sheffield Hallam University
Nottingham Trent University
Julian Briggs &
Richard Hibberd
For help type (help)

3.2
zal: (compose |
```

```
Zal Function COMPOSE xr yr   Compose. Composes 2 relations, yr & xr. The range of yr must be a subset of the domain of xr. Returns a relation which maps the domain of yr to the range of xr.
Method: 1. Remove 'r's. 2. Replace the second of each maplet (snd) in yr by xr applied to snd. 3. Replace the 'r.   Eg (compose [#('a 'ant) #('b 'ball)] [#(1 'a) #(2 'b)]) ![#(1 ANT) #(2 BALL)].   Eg
(applyz (compose [#('a 'ant) #('b 'ball)][#(1 'a) #(2 'b)])1) !ANT.
```

Figure 5-19 : Documentation Strings

When the name of a data object is typed, the current binding is displayed in the status bar, though in its internal representation.


Allegro CL 3.0.2 [e:\acl302\allegro.img]
File Edit Search Window Package Tools Builder
Preferences Help
Toploop Package: zal

```
Z Animation in Lisp
Sheffield Hallam University
Nottingham Trent University
Julian Briggs &
Richard Hibberd
For help type (help)

3.2
zal: (compose Cars
```

```
Zal Variable   CARS   is bound to the cons   (*S (C1 PONTIAC) (C2 MINI) (C3 SKODA) (C4
SKODA) (C5 FIAT) (C6 SKODA))
```

Figure 5-20 : Feedback via the Status Bar

5.3.1 Discussion

As was described in the introduction, this study is unusual in that the animator and the ZAL notation were used as a prototyping specification language. In the particular situation where this arose, this approach was adopted due to a lack of familiarity with the Z notation and the exploratory interaction with the animator was used to establish correct behaviour; this ZAL representation was 'reverse-engineered' to give a version in Z notation. While the lack of rigour of this approach precludes any great confidence in the final Z version, as is evidenced by the errors therein, the expedient of correcting these and transforming that Z version into ZAL and *validating using the derived version* would have re-established the correspondence and the confidence. Had this been done, all the claimed benefits of the more usual style would have been available – the process could have been rationalised as a minor modification to the REALiZE process, wherein the initial attempt at formalisation is made in ZAL notation, which is manually translated into Z.

5.4 Summary

From the three widely differing case studies, it can be seen that the ZAL execution engine has the capacity to animate and thereby to demonstrate a significant proportion of the Z language. The variety of the problem domains and also of the styles of use attest to the robust and flexible functionality it offers a user. Even cursory examination of the Z that is being evaluated will confirm that complex expressions pose no problems.

The ability to promote schema execution outcomes allows a sequence of operations to be explored in a structured and coherent way; the interactive interface can be used to examine both the state elements and the operations upon them. This interaction can be used to assist in the development of a specification, as in the second and third studies, or to demonstrate an existing specification, as in the first. Both of these styles can be accommodated within the framework of the REALiZE process.

The availability of a scripting interface to the animator can support 'downstream' Software Engineering activities, in particular the creation and management of testing suites which could then be administered to subsequent releases of software. While this would be of value, the provision of the schema composition and schema piping operators offers the possibility of the automated testing of specifications.

6 Results and Conclusions

It can be seen from Section 5 that ZAL provides robust and flexible functionality, as demonstrated in a variety of domains and with very different styles of Z specifications; the coverage of individual Z expressions is extensive which with the schema calculus operations provides for the animation of complex specifications containing complicated expressions.

It is not enough however to present the developed software and judge it simply on its functionality; it must be evaluated in the light of the original research objectives and against the questions that were posited at that stage. These will be addressed in turn, but the original objective of

> *"the provision of Requirements Engineering ...tools that assist in representing, validating and evolving requirements so as to deliver a high quality requirements document ..."*

must be judged to have been met; the combination of TranZit, to capture and formalise a specification, and ZAL, to then animate and validate it, do constitute a toolset that precisely meets this objective.

6.1 Results evaluated

With reference to the issues more specific to this work, two Requirements Engineering activities have been shown to be supported explicitly by the use of the ZAL animator :

- the requirements formalisation process where the *writer* of a specification uses the animator to confirm that the Z document correctly captures the behaviour that *the writer intends*;
- the requirements validation process where, having written the specification, the writer demonstrates the specification to the client or sponsor, to establish that the Z document correctly captures the behaviour that *the sponsor intends.*

Both of these processes are obviously fundamental to the production of a high quality requirements document, and this is the basis for the claim of success

The questions from Section 1.4 are reiterated here:

- *what form might such an animation system take and how would animation be incorporated into the requirements capture process?*

The form of the system developed organically, consistent with the exploratory programming approach adopted; the current evolution, version 3.2, comprises a robust "execution engine" that evaluates expressions formed in the extended Lisp notation of the ZAL language, together with a graphical interface that allows the symbolic execution of Z schemas that have been translated into ZAL notation by another toolset component, named TranZit. An adjunct of the conventional development lifecycle, called the REALiZE method, has been proposed, to incorporate validation by animation; though not part of this particular research programme, this is the context within which it has been developed.

- *which aspects of the process might animation help?*

The support of requirements validation is explicitly supported by the toolset component that has been developed, but the overall process can be considered to have been enhanced by the ability to address and perhaps resolve the fundamental difficulty of the use of a formal specification notation – that of accessibility to and ownership of the formal document **by the project sponsor and other stakeholders, regardless of their experience of reading and understanding Z specifications.**

Furthermore, the animation can also be used by the specification writer to verify that the Z that he has written is in fact that which he intended.

- *can animation facilities be provided that are sufficiently usable – i.e. that will enhance rather than simply complicate the process?*

Evaluation of usability can be fraught with subjective assessments, rather than objective metrics; the description of the provision as a *developer's interface* recognises that the ZAL environment will be used by the specifier, rather than any other team member. This requires expertise in the use of the animator on the part of only those skilled in the writing of Z specifications; it is expected that the interface provided will be natural and easy to learn for individuals with this background. An alternative view of the animation execution process is that of the animation system providing a richer environment for the dialogue between the participants; in this environment, the subtleties of understanding can be jointly explored and resolved.

- *can the inductive style of specification development be supported, in contrast to the more usual post hoc deductive proof offered by other tools?*

This is perhaps more straightforward, insofar as there is no alternative; a deductive validation of requirements **with respect to user intention** is not possible, so support for inductive development is all that can be offered.

- *what coverage of the Z specification language might such an animation system provide and how would the system be developed?*

The animation system has been developed incrementally, particularly the provision of expression level functionality; the coverage is significant but not complete, though the implementation of further constructs at the expression level remains relatively straightforward. The areas where further development is more useful and also more problematic are considered in Section 6.2 *Future work*

6.2 Future work

The generally positive response that this project has generated thus far from the research community indicates that further work may be appropriate; the debate regarding the value of the execution of specifications may not be won, but the current popularity of animators suggests that the tide has turned.

The outstanding issue to be addressed in this project is that of extending the schema calculus functionality, primarily to support schema composition and piping. This is a question of modelling non-determinism and the mechanism illustrated in `non-deterministic-pick` (Section 4.6) offers a way forward.

The provision of these schema operators would allow the development of automated test administration using the scripting interface as suggested in Section 2.4.2.

Development in an altogether different direction is also possible; the robust engine at the heart of the ZAL animator could be used as the basis for a Z desktop calculator, as suggested by [Utting 2000]. This would require a different interface, but could support the inductive development of expressions in the way that the command line was used in the Car Rental case study (Section 5.3).

6.3 In conclusion

The method proposed has facilitated the creation of a precise, concise, unambiguous statement of requirements, the essential deliverable of the formal specification process; it was arrived at by a largely inductive, consistent with the original premise and as it is a formal document, it is still available for an implementation to be proved against. That elements of the specification may have been "reverse-engineered" after the desired behaviour had been elicited by animation does not devalue it in the least; there is a famous precedent that suggests that even it has not been generated by a wholly rational, prescribed process, it is worthwhile to "pretend" that it had, which is "The Rational

Design Process: How and Why to Fake It by David Parnas and Paul Clements [Parnas and Clements 1986].

It is possible to claim success in this endeavour, since it has been demonstrated that:

the specification developed used the ZAL animator within the REALiZE process is better because it is better validated and because the animation process has provided *accessibility* to the specification and has thereby generated wider ownership;

the ability to maintain the (largely unconstrained) Z specification as the medium of development and refinement while providing the facility to animate and interact with the automatically derived ZAL equivalent addresses and indeed resolves many of the concerns and difficulties of animation.

References

[Abelson *et al* 1985] Abelson H, Sussman G & Sussman J, *"Structure and Interpretation of Computer Programs"*, MIT Press, 1995

[Abrial 1980] Abrial J, *"The Specification Language Z: basic library"*, Oxford University Programming Research Group, 1998

[Alford and Lawson 1979] Alford M & Lawson J, *"Software Requirements Engineering Methodology (Development)"*, RADC-TR-79-168, USAF Rome Air Development Centre, New York, 1979

[Andersen *et al* 1992] Andersen M Elmstrom R Lassen P & Larsen P, *"Making Specifications Executable - Using IPTES Meta-IV"*, Microprocessing & Microprogramming, Vol. 35, No. 1992

[Backus 1978] Backus J. *"Can programming be liberated from the von Neumann style? A functional style and its algebra of programs."* Communications of ACM August 1978

[Barden *et al*] Barden S Stepney S & Cooper D, *"Z in Practice"*, Prentice Hall International (UK), 1994

[Basili and Perricone 1984]Basili V & Perricone B, *"Software errors and complexity: An empirical investigation"*, Communications of the ACM, 27(1), 1984

[Breuer and Bowen 1994] Breuer P & Bowen J, *"Towards Correct Executable Semantics for Z"*, in Bowen J and Hall J (eds.) Z User Workshop, Cambridge 1994, Workshops in Computing, Springer-Verlag, 1994.

[Brooks 1987] Brooks Jr. F P, "No Silver Bullet: Essence and Accidents of Software Engineering",IEEE Computer, 1987

[Buckberry 1999] Buckberry G, *"An Editor and Transformation System for a Z Animation Case Too"*, PhD thesis, Sheffield Hallam University, UK, 1999

[DeMarco and Lister 1989] DeMarco T & Lister T, Software Development: State of the Art vs. State of the Practice in Proceedings 11th International Conference on Software Engineering, IEEE, 1989

[Dick *et al* 1990] Dick A Krause J & Cozens J, *"Computer Aided Transformation of Z into Prolog"* in Proceedings of the Fourth Annual Z User Meeting, Z User Workshop, Nicholls J (ed), Springer-Verlag Germany, 1990

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

[Forberg and Mooz 1997] Forberg K & Mooz H, *"System Engineering Overview"*, Software Requirements Engineering, 2nd ed, Thayer R H & Dorfman M, Eds, IEEE Computer Society Press, Los Alamitos USA, 1997

[Fuchs 1992] Fuchs N, *"Specifications are (preferably) Executable"*, Software Engineering Journal, September 1992

[Gibbs 1994] Gibbs, W.W. ``Software's Chronic Crisis." Scientific American, September, 1994.

[Glaser *et al*1984] Glaser H Hankin C & Till D, *"Principles of Functional Programming"*, Prentice Hall International, 1984

[Goodman 1993] Goodman H, *"Animating Z specifications in Haskell using a monad"*, Technical Report, University of Birmingham UK, 1993

[Graham 1994] Graham P, *"On Lisp"*, Prentice Hall, 1994

[Harel 1987] Harel D., *"Algorithmics : the spirit of computing"*, Addison-Wesley, 1987

[Harvey & Wright 1994] Harvey B. & Wright M., *"Simply Scheme"*, , MIT Press, 1994

[Hayes 1993] Hayes I (ed), *"Specification Case Studies"*, Prentice Hall 2nd edition, 1993

[Hayes and Jones 1989] Hayes I & Jones C, *"Specifications are not (necessarily) executable"*, Software Engineering Journal, November 1989

[Hibberd 1990] Hibberd R, *"An FP System"*, MSc Project Sheffield Hallam University, 1990

[Hörcher 1994] Hörcher H-M, *"Animation and Prototyping of implicit Specifications"*, Technical Report, DST Deutsche System-Technik GmbH, Germany, 1994

[IEEE 1993] *"IEEE Recommended Practice for Software Requirements Engineering"*, IEEE Std 830, 1993

[ISO/IEC 14882 1998] *"Information Technology - Programming Languages - C++"*, ISO, 1998

[Jackson and Stokes 1993], *"Formal specification and animation of of a water level monitoring system."*, Research Report INFO-0428. Ottawa, Atomic Energy Control Board, 1993

[Johnson and Sanders 1989] Johnson M & Sanders P, *"From Z Specification to Functional Implementations"* in Proceedings of the Z User Workshop Oxford 1989, Nicholls J (ed), Springer-Verlag, Germany, 1989

[Jones 1990] Jones C, *"Systematic Software Development using VDM"*, Prentic Hall International, New Jersey, 1990

[Keene 1989] Keene S, *"Object-Oriented Programming in Common Lisp : A Programmer's Guide to CLOS"*, Addison-Wesley, 1989

[Leveson1990] Leveson N, *"Guest Editor's Introduction: Formal Methods in Software Engineering"*, IEEE Transactions on Software Engineering 16 (9), 1990

[Lightfoot 1991] Lightfoot D, *"Formal Specification Using Z"*, Macmillan Press, 1991

[Main Commission Aircraft Accident Investigation 1994] *"Report on the Accident to Airbus A320-211 Aircraft in Warsaw on 14 September 1993"*, Main Commission Aircraft Accident Investigation 1994

[McCarthy 1960] McCarthy J, *"Recursive Functions of Symbolic expressions and their Computation by Machine, Part 1"*, Communications of the ACM 3(4), 1960

[Meyer 1985] Meyer B, *"On Formalism in Specifications"*, IEEE Software 2(1), 1985

[Mullery 1996] Mullery *"The Perfect Requirement Myth"*, Requirements Engineering Journal (1996) 1:132-134 Springer-Verlag London, 1996

[Page *et al* 1993]Page D, Williams P & Boyd D, *"Report of the Inquiry into London Ambulance Service"*, South West Thames Regional Health Authority, 1993

[Parnas and Clements 1986]Parnas d & Clements P, *"The Rational Design Process: How and Why to Fake It"*, IEEE Transactions on Software Engineering, Vol. SE12 No 2, February 1986

[Parry 2001] Parry P, *"Viz"* - work in progress, Sheffield Hallam University, 2001

[Perry and Steig 1993] Perry D & Steig C, *"Investigating software requirements errors"*, in Proceedings of th eEuropean Software Engineering Conference, 1993

[PeytonJones and Hughes 1998] Peyton Jones S & Hughes J (eds), *"Report on the Programming Language Haskell 98"*, http://www.haskel.org/report

[Pfleeger 1998] Pfleeger S, *"Software Engineering: Theory and Practice"*, Prentice Hall, New Jersey, 1998

[Pile 1991],Pile I, *"Developing Safety Systems"*, Prentice Hall, 1991

[Potter *et al* 1991] Potter B Sinclair J & Till D, *"An Introduction to Formal Specification and Z"*, Prentice Hall International (UK), 1991

[Runciman and Wakeling 1995] Runciman C & Wakeling D , *"Applications of Functional Programming"*, UCL Press, 1995

[Sherrill and Carver 1993] Sherrell L.B. & Carver D.L., *"Z Meets Haskell: A Case Study"*, COMPSAC '93 - Procs. 17th. Annual International Computer Software & Applications Conference,1993

[Siddiqi *et al* 1998] Siddiqi J Morrey I Hibberd R & Buckberry G,*" Understanding and exploring formal specifications"*, Annals of Software Engineering 6 (1/4), 1998

[Somerville 1989]] Somerville I, *"Software Engineering (3rd Ed.)"*; Addison Wesley, 1989

[Somerville and Sawyer 1997] Somerville I & Sawyer P,*"Requirements Engineering: A Good Practice Guide"*, Wiley, 1997

[Spivey 1992] Spivey J.M., The Z Notation A Reference Manual, Prentice Hall, 1992

[Steele and Sussman 1975] Steele G & Sussman G, *"Scheme: An interpreter for the extended lambda calculus"*, Memo 349, MIT artificial Intelligence Laboratory

[Stepanov and Lee 1994] Stapanov A & Lee M, *"The Standard Template Library"*, Hewlett Packard Company, Palo Alto USA, 1994

[Utting 2000] Utting M, *"Data Structures for Z Testing Tools"* presented at FM-TOOLS 2000, University of Ulm Germany, July 2000

[Valentine 1991] Valentine S, *"Z--, An Executable Subset of Z"*, 6th Annual Z User Meeting, York UK, 1991

[Valentine 1993] Valentine S, *"Putting Numbers into the Mathematical Toolkit"*, Proceedings of theSeventh Z USer Meeting December 1992, in Bowen and Nicholls (eds) Springer-Verlag, 1993

[van Schouwen] van Schouwen A, *"The A-7 requirements model: re-examination for real-time systems and an application to monitoring systems"*, Technical report 90-276, Queens University, Kingston, Ontario, 1991 [van Vliet 2000] van Vliet H, *"Software Engineering Principles and Practice"*, John Wiley, New York, 2000

[West and Eaglestone 1992] West M & Eaglestone B, *"Software development: two approaches to animation of Z specifications using Prolog"*, Software Engineering Journal, July 1992

[Williams 1994] Williams L, *"Assessment of safety-critical specifications"*, IEEE Software, 11(1), pp.51-59, 1994

[Wordsworth 1992] Wordsworth J, *"Software Development with Z : A Practical Approach to Formal Methods in Software Engineering"*, Addison-Wesley, 1992

Appendix A (STL/C++ structures)

```
#include <map>
#include <set>
#include <algorithm>

template <class DT, class RT>
set <DT>  dom( const multimap<DT, RT> & mapp)
{
     set<DT> result;
     for (multimap<DT, RT>::const_iterator mi = mapp.begin();
mi!=mapp.end();mi++)
        result.insert( mi->first);
     return result;
}

template <class DT, class RT>
set <RT>  ran( const multimap<DT, RT> & mapp)
{
     set<RT> result;
     for (multimap<DT, RT>::const_iterator mi = mapp.begin();
mi!=mapp.end();mi++)
        result.insert( mi->second);
     return result;
}

template <class DT, class RT>
multimap<DT, RT>  domRes( const set<DT> & sett, const multimap<DT, RT> & mmapp)
{
     multimap<DT, RT> result;
     pair<multimap<DT, RT>::const_iterator ,multimap<DT, RT>::const_iterator >
pr;
     for (set<RT>::const_iterator si = sett.begin(); si != sett.end(); si++)
     {
       pr = mmapp.equal_range( *si);
       copy(pr.first, pr.second, inserter(result,result.begin()));
     }
     return result;
}

template <class DT, class RT>
set< RT>  relImage( const multimap<DT, RT> & mmapp,  const set<DT> & sett)
{
     return ran(domRes(sett, mmapp));
}

template <class DT, class RT>
multimap<DT, RT>  ranRes(  const multimap<DT, RT> & mmapp, const set<RT> & sett)
{
     return inverse(domRes(sett, inverse(mmapp)));
}
```

```
template <class DT, class RT>
multimap<RT, DT>  inverse(  const multimap<DT, RT> & mmapp)
{
     multimap<RT, DT> result;
     for (multimap<DT, RT>::const_iterator mi = mmapp.begin();
mi!=mmapp.end();mi++)
        result.insert(make_pair(mi->second, mi->first));
     return result;
}

template <class ST >
set<ST> inter(const set<ST>  & s1,const set<ST> & s2)
{
     set<ST> result;
     set_intersection (s1.begin(), s1.end(),s2.begin(), s2.end(),
inserter(result, result.begin()));
     return result;
}

template <class ST >
set<ST> sunion(const set<ST>  & s1, const set<ST> & s2)
{
     set<ST> result;
     set_union (s1.begin(), s1.end(), s2.begin(), s2.end(),
                                              inserter(result, result.begin()));
     return result;
}
```

Appendix B (Car Rental Specification)

BSc SE/CM/CMS Final Year

Formal Software Development Assessment (Part 1)

This is GROUP work

Write a Z specification for the system informally described below.

The ACME car hire company has a fleet of cars distributed across the country in a number of depots in several major cities. Each car is identified by a unique id number. The depot in which the car is currently garaged, the car's current mileage and the car's manufacturer is recorded for each car. A client can hire a car from any depot and return it to any other depot. When a customer hires a car from a particular depot, they provide their name and specify the make of car they want, and the hire date is recorded. A specific car (if available) is then allocated to them. When the car is returned (possibly to another depot), its new mileage is inspected, and the customer is charged 10p per mile plus a fixed charge of £20 for each day of the hire period. The ACME company is so successful that sometimes there's a queue of people waiting to return their hired car to a depot. When this happens, the company deals with the queue in strict order. Occasionally a customer in the queue gets tired of waiting and leaves, hoping to return later when the queue is shorter.

Write a Z specification for the system informally described above. You should include specifications for the following operations:

(a) Hiring a car. (This operation ignores cars which may be in the returning queues - only those cars actually in the depots are available for hire.)

(b) Returning a car. (This comprises essentially two operations - joining a particular returning queue, and dealing with the car at the front of a queue by issuing a bill etc.)

(c) Answering the following queries:

(i) Which depots currently have a car of this particular make?

(ii) Which cars are currently garaged at this particular depot?

(iii) Is there a returning car of this particular make queueing at this particular depot? If so, what position is it in the queue?

State clearly any assumptions you make, and remember that the English commentary component of a Z specification is an important aid to understanding, so please include it.

Appendix C (WLMS in Z)

/***

Case Study : Water Level Monitoring System in Z

From : A. Van Shouwen

Date : 5th December 1997

Informal Description :

This specification is concerns the operation of a Water Level Monitoring system which might be used in a safety-critical system involved in steam generation, for example in a power plant. The system consists of two reservoirs; one serving as a steam generation vessel, and the other as a source of water. Under normal operation, water is pumped from the source into the steam generating vessel where it is evaporated. The pump transfering water to the generating vessel and the pump controlling the rate of steam generation in regulated by a control system termed the WLMS.

The WLMS monitors and displays the level of water in the stream generating vessel. When the water level is too high or low, the WLMS issues visible and audible alarms and shuts down the pumps. Pumps are also shut down if the WLMS itself fails either due to external faults (such as failure of the water level detector) or internal faults in the WLMS computer. Internal faults are detected by an external watchdog which receives a periodic KICK from the WLMS. If and external faults is detected by the WLMS or the watchdog fires, the WLMS shuts the system down by turning off power to both pumps.

In addition, the WLMS has two push buttons: Selftest lets the operator test the WLMS output hardware whilst the system is shut down. Reset returns the system to normal operation following shutdown or test, provided that the water level is within the specified limits.

****

/* Formal Specification in Z */

```
BYTE  ==     0..255

TIME  ==     ℕ

LEVEL ==     ℕ

DEVICETYPE ::= ok | failed

WATCHDOGTYPE ::= uninit | operate | shut

ONOFFTYPE ::= on | off

BUTTONTYPE ::= pressed | released

OPERATINGMODETYPE ::= operating | shutdown | standby | test

FAILUREMODETYPE ::= allok | badlevdev | hardfail

PUMPSWITCHTYPE ::= open | closed

SHUTDOWNSIGNALTYPE ::= go | stop

ALARMTYPE ::= silent | audible | undefined
```

```
┌StoredData──────────────────────────────────────
│time : TIME
│timeInMode : TIME
│watchDogTime : TIME
│resetButtonTime : TIME
│selftestButtonTime : TIME
│operatingMode : OPERATINGMODETYPE
│failureMode : FAILUREMODETYPE
│alarm : ALARMTYPE
│shutdownSignal : SHUTDOWNSIGNALTYPE
│waterlevel : LEVEL
│level : LEVEL
│step : TIME
│
└────────────────────────────────────────────────
```

```
┌ControlSignals──────────────────────────────────
│alarm : ALARMTYPE
│shutdownSignal : SHUTDOWNSIGNALTYPE
│pumpSwitch : PUMPSWITCHTYPE
│watchdog : WATCHDOGTYPE
│
└────────────────────────────────────────────────
```

```
┌MonitorVar──────────────────────────────
│diffPress : BYTE
│resetButton : BUTTONTYPE
│selftestButton : BUTTONTYPE
│powerNow : ONOFFTYPE
│memory : DEVICETYPE
│timeNow : TIME
│timeDevice : DEVICETYPE
└────────────────────────────────────────
```

```
┌
│levelLowerCal : LEVEL
│levelUpperCal : LEVEL
│shutdownLockTime : TIME
│watchdogtimeout : TIME
│hysteresis : LEVEL
│highWaterLimit : LEVEL
│initTime : TIME
│lowWaterLimit : LEVEL
│maxAlarmTime : TIME
│maxSelftestDelay : TIME
│maxResetDelay : TIME
│maxTestDelay : TIME
├────────────────────────────────
│levelLowerCal = 130
│levelUpperCal = 270
│shutdownLockTime = 200
│watchdogtimeout = 500
│hysteresis = 5
│highWaterLimit = 260
│lowWaterLimit = 140
│maxAlarmTime = 4000
│maxSelftestDelay = 500
│maxResetDelay = 3000
│maxTestDelay = 14000
└
```

```
┌AlarmControl──────────────────────────
│MonitorVar?
│ΔStoredData
├──────────────────────────
│alarm' =
│      if powerNow? ≠ on ∨ failureMode = hardfail
│      then undefined
│      else   if failureMode = badlevdev
│             then    if timeInMode = 0 then audible else alarm
│             else    if operatingMode = test
│                     then    if 0 ≤ timeInMode < maxAlarmTime then audible else silent
│                     else    if operatingMode = standby
│                             then alarm
│                             else    if lowWaterLimit < waterlevel < highWaterLimit
│                                     then    if operatingMode = shutdown
│                                             then alarm
│                                             else    if timeInMode = 0
│                                                     then silent
│                                                     else alarm
│                                     else audible
└──────────────────────────
```

```
┌Initialise──────────────────────────
│StoredData
├──────────────────────────
│0 ≤ initTime ≤ 5000
│alarm = silent
│shutdownSignal = stop
│timeInMode = 0
│watchDogTime = 0
│resetButtonTime = 0
│selftestButtonTime = 0
│operatingMode = standby
│failureMode = allok
│0 ≤ time ≤ initTime
└──────────────────────────
```

```
┌GetNextMode───────────────────────────
│Monitor Var?
│ΔStoredData
├──────────────────────────────────────
│operatingMode' =
│     if operatingMode = operating
│     then if selftestButton? ≠ pressed ∧ ¬ (lowWaterLimit < waterlevel < highWaterLimit)
│          then shutdown
│          else     if selftestButton? = pressed ∧ selftestButtonTime ≥ maxSelftestDelay
│                   then test
│                   else operating
│     else if operatingMode = shutdown
│        then if selftestButton? ≠ pressed ∧ timeInMode < shutdownLockTime ∧
│            (lowWaterLimit + hysteresis < waterlevel < highWaterLimit - hysteresis)
│                   then operating
│                   else     if selftestButton? ≠ pressed ∧
│                               timeInMode ≥ shutdownLockTime
│                            then standby
│                            else     if selftestButton? = pressed ∧
│                                     selftestButtonTime ≥ maxSelftestDelay
│                                     then test
│                                     else shutdown
│             else     if operatingMode = standby
│                      then     if resetButtonTime > maxResetDelay ∧
│                               (lowWaterLimit + hysteresis < waterlevel
│                               < highWaterLimit - hysteresis)
│                               then operating
│                               else if selftestButton? = pressed ∧
│                                       selftestButtonTime ≥ maxSelftestDelay
│                                       then test
│                                       else standby
│                      else     if timeInMode ≥ maxTestDelay
│                               then standby
│                               else test
└──────────────────────────────────────
```

```
NormalOperation
MonitorVar?
ΔStoredData
ControlSignals!
GetNextMode
AlarmControl
────────────────────────────
time' = timeNow?
 level = levelLowerCal +
      ( (diffPress? * 103803 - 485010) * (levelUpperCal - levelLowerCal) ) div 2550000
      waterlevel =
              if diffPress? = 255 then levelLowerCal - 1
              else if diffPress? = 0 then levelUpperCal + 1
              else if level < levelLowerCal then levelLowerCal
              else if level > levelUpperCal then levelUpperCal
              else level
watchdog! = if watchDogTime < watchdogtimeout then operate else shut
step = timeNow? - time
watchDogTime' = step
resetButtonTime' = if resetButton? = pressed then resetButtonTime + step else 0
selftestButtonTime' = if selftestButton? = pressed then selftestButtonTime + step else 0
timeInMode' = if operatingMode = operatingMode' then timeInMode + step else 0
failureMode' =
      if failureMode = hardfail ∨ memory? ≠ ok ∨ watchdog! ≠ operate
        ∨ timeDevice? = failed
      then hardfail
      else    if failureMode = badlevdev ∨ diffPress? ∈ { 0, 255 }
              then badlevdev
              else allok
shutdownSignal' =
      if failureMode ∈ { badlevdev, hardfail } ∨ operatingMode ∈ { standby, test }
      then stop
      else if operatingMode = shutdown then shutdownSignal
      else if resetButton? = pressed then shutdownSignal
      else go
pumpSwitch! =
      if powerNow? = on ∧ shutdownSignal' = go ∧ watchdog! = operate then closed
      else open
shutdownSignal! = shutdownSignal'
alarm! = alarm'
```

Appendix D (WLMS results)

Validation of safety properties by animation December 17, 1999							
Table 1. Results of the animation.							
Inputs:				state variables:			
time button	*reset button*	*test level*	*water mode*	*operating mode*	*failure*	*alarm*	*pumps*
2000				*standby*	*allok*	*silent*	*stop*
2050	*released*	*released*	*normal*	*standby*	*allok*	*silent*	*stop*
2100	*pressed*	*released*	*normal*	*standby*	*allok*	*silent*	*stop*
3100	*pressed*	*released*	*normal*	*standby*	*allok*	*silent*	*stop*
3500	*pressed*	*released*	*toolow*	*standby*	*allok*	*silent*	*stop*
4100	*pressed*	*released*	*low*	*standby*	*allok*	*silent*	*stop*
4800	*pressed*	*released*	*normal*	*standby*	*allok*	*silent*	*stop*
5099	*pressed*	*released*	*normal*	*standby*	*allok*	*silent*	*stop*
5100	*pressed*	*released*	*normal*	*operating*	*allok*	*silent*	*stop*
5200	*released*	*released*	*normal*	*operating*	*allok*	*silent*	*go*
5600	*pressed*	*released*	*normal*	*operating*	*allok*	*silent*	*go*
6700	*pressed*	*released*	*high*	*operating*	*allok*	*silent*	*go*
6750	*pressed*	*released*	*toohigh*	*shutdown*	*allok*	*audible*	*go*
6820	*pressed*	*released*	*high*	*shutdown*	*allok*	*audible*	*go*
7000	*pressed*	*released*	*normal*	*operating*	*allok*	*audible*	*go*
7400	*pressed*	*released*	*normal*	*operating*	*allok*	*silent*	*go*
8700	*pressed*	*released*	*normal*	*operating*	*allok*	*silent*	*go*
8800	*released*	*released*	*normal*	*operating*	*allok*	*silent*	*go*
8820	*released*	*released*	*low*	*operating*	*allok*	*silent*	*go*
8830	*released*	*released*	*low*	*operating*	*allok*	*silent*	*go*
8840	*released*	*released*	*toolow*	*shutdown*	*allok*	*audible*	*go*
8880	*released*	*released*	*low*	*shutdown*	*allok*	*audible*	*go*
8900	*released*	*released*	*normal*	*operating*	*allok*	*audible*	*go*

Appendix D Table 1

Validation of safety properties by animation December 17, 1999							
Table 2. Results of the animation.							
Inputs:				state variables:			
time button	*reset button*	*test level*	*water mode*	*operating mode*	*failure*	*alarm*	*pumps*
8900				*operating*	*allok*	*audible*	*go*
9000	*released*	*pressed*	*normal*	*operating*	*allok*	*silent*	*go*
9040	*pressed*	*pressed*	*high*	*operating*	*allok*	*silent*	*go*
9070	*pressed*	*pressed*	*toohigh*	*operating*	*allok*	*audible*	go
9080	*pressed*	*pressed*	*toohigh*	*operating*	*allok*	*audible*	go
9090	*released*	*pressed*	*toohigh*	*operating*	*allok*	*audible*	go
9300	*released*	*pressed*	*normal*	*operating*	*allok*	*audible*	*go*
9499	*released*	*pressed*	*normal*	*operating*	*allok*	*audible*	*go*
9500	*released*	*pressed*	*normal*	*Test*	*allok*	*audible*	*go*
9501	*released*	*pressed*	*normal*	*Test*	*allok*	*audible*	*stop*
9510	*released*	*released*	*normal*	*Test*	*allok*	*audible*	*stop*
10700	*pressed*	*released*	*normal*	*Test*	*allok*	*audible*	*stop*
11100	*pressed*	*pressed*	*normal*	*Test*	*allok*	*audible*	*stop*
11920	*released*	*pressed*	*normal*	*Test*	*allok*	*audible*	*stop*
12300	*released*	*released*	*normal*	*Test*	*allok*	*audible*	*stop*
12750	*released*	*released*	*toolow*	*Test*	*allok*	*audible*	stop
13249	*released*	*released*	*normal*	*Test*	*allok*	*audible*	*stop*
13500	*released*	*released*	*normal*	*Test*	*allok*	*audible*	*stop*
13999	*released*	*released*	*normal*	*Test*	*allok*	*silent*	*stop*
14500	*released*	*released*	*normal*	*Test*	*allok*	*silent*	*stop*
14600	*pressed*	*released*	*normal*	*standby*	*allok*	*silent*	*stop*

Appendix D Table 2

Validation of safety properties by animation December 17, 1999							
Table 3. Results of the animation.							
Inputs:				state variables:			
Time button	*reset button*	*test level*	*water mode*	*operating mode*	*failure*	*alarm*	*pumps*
14600				*standby*	*allok*	*silent*	*stop*
15000	*pressed*	*released*	*normal*	*standby*	*allok*	*silent*	*stop*
17300	*pressed*	*released*	*normal*	*standby*	*allok*	*silent*	*stop*
17600	*pressed*	*released*	*toohigh*	*standby*	*allok*	*silent*	*stop*
17610	*pressed*	*released*	*high*	*standby*	*allok*	*silent*	*stop*
17620	*pressed*	*released*	*normal*	*operating*	*allok*	*silent*	*stop*
17640	*released*	*released*	*high*	*operating*	*allok*	*silent*	*go*
17650	*released*	*released*	*toohigh*	*shutdown*	*allok*	*audible*	*go*
17850	*released*	*released*	*high*	*shutdown*	*allok*	*audible*	*go*
17860	*released*	*released*	*low*	*standby*	*allok*	*audible*	*go*
17870	*released*	*released*	*toolow*	*standby*	*allok*	*audible*	*stop*
17900	*released*	*released*	*low*	*standby*	*allok*	*audible*	*stop*
17950	*released*	*released*	*normal*	*standby*	*allok*	*audible*	*stop*
18000	*released*	*pressed*	*normal*	*standby*	*allok*	*audible*	*stop*
18150	*released*	*pressed*	*normal*	*standby*	*allok*	*audible*	*stop*
18170	*released*	*pressed*	*low*	*standby*	*allok*	*audible*	*stop*
18180	*released*	*pressed*	*toolow*	*standby*	*allok*	*audible*	*stop*
18200	*released*	*pressed*	*low*	*standby*	*allok*	*audible*	*stop*
18499	*released*	*pressed*	*normal*	*standby*	*allok*	*audible*	*stop*
18500	*released*	*pressed*	*normal*	*Test*	*allok*	*audible*	*stop*
18600	*released*	*pressed*	*normal*	*Test*	*allok*	*audible*	*stop*
18700	*released*	*released*	*normal*	*Test*	*allok*	*audible*	*stop*
19550	*released*	*released*	*normal*	*Test*	*allok*	*audible*	*stop*
19570	*released*	*released*	*low*	*Test*	*allok*	*audible*	*stop*
19590	*released*	*released*	*toolow*	*Test*	*allok*	*audible*	*stop*
19600	*released*	*released*	*low*	*Test*	*allok*	*audible*	*stop*
19700	*released*	*released*	*normal*	*Test*	*allok*	*audible*	*stop*
22600	*released*	*released*	*normal*	*Test*	*allok*	*audible*	*stop*
23000	*released*	*released*	*normal*	*Test*	*allok*	*silent*	*stop*
23600	*released*	*released*	*normal*	*Test*	*allok*	*silent*	*stop*
24000	*released*	*released*	*normal*	*standby*	*allok*	*silent*	*stop*

Appendix D Table 3

Validation of safety properties by animation December 17, 1999							
Table 4. Results of the animation.							
Inputs:				state variables:			
time button	*reset button*	*test level*	*water mode*	*operating mode*	*failure*	*alarm*	*pumps*
24000	*standby*	*allok*	*silent*	*stop*			
24400	*released*	*released*	*normal*	*standby*	*allok*	*silent*	*stop*
25000	*pressed*	*released*	*normal*	*standby*	*allok*	*silent*	*stop*
28000	*pressed*	*released*	*normal*	*operating*	*allok*	*silent*	*stop*
28600	*released*	*released*	*toolow*	*shutdown*	*allok*	*audible*	*go*
28760	*released*	*pressed*	*toolow*	*shutdown*	*allok*	*audible*	*go*
29000	*released*	*pressed*	*toolow*	*shutdown*	*allok*	*audible*	*go*
29100	*released*	*pressed*	*normal*	*shutdown*	*allok*	*audible*	*go*
29260	*released*	*pressed*	*normal*	*test*	*allok*	*audible*	*go*
29300	*released*	*pressed*	*normal*	*test*	*allok*	*audible*	*stop*
31700	*pressed*	*pressed*	*normal*	*test*	*allok*	*audible*	*stop*
32000	*released*	*released*	*normal*	*test*	*allok*	*audible*	*stop*
33700	*released*	*released*	*normal*	*test*	*allok*	*silent*	*stop*
34900	*released*	*released*	*toohigh*	*standby*	*allok*	*silent*	*stop*
34901	*released*	*released*	*faulty*	*standby*	*badlevdev*	*silent*	*stop*
34903	*released*	*released*	*normal*	*standby*	*badlevdev*	*silent*	*stop*
37100	*pressed*	*released*	*normal*	*standby*	*hardfail*	*undefined*	*stop*
40100	*pressed*	*released*	*normal*	*standby*	*hardfail*	*undefined*	*stop*
40101	*pressed*	*released*	*normal*	*operating*	*hardfail*	*undefined*	*stop*
40200	*released*	*released*	*normal*	*operating*	*hardfail*	*undefined*	*stop*
40500	*released*	*released*	*normal*	*operating*	*hardfail*	*undefined*	*stop*

Appendix D Table 4

Appendix E (choose)

```
(in-package 'zal)
;; non-deterministic choice, as modeled by P Graham, "On Lisp", Prentice Hall, 1994
(defvar choose nil)

(setq *cont* #'identity)

(defmacro =lambda (parms &body body)
   `#'(lambda (*cont* ,@parms) ,@body))



(defmacro =defun (name parms &body body)
   (let
        ((f (intern (concatenate 'string "=" (symbol-name name)))))
      `(progn
         (defmacro ,name , parms
            `(,',f *cont* ,,@parms))
         (defun ,f (*cont* ,@parms) ,@body))))

(defmacro =bind (parms expr &body body)
   `(let ((*cont* #'(lambda ,parms ,@body))) ,expr))

(defmacro =values (&rest retvals)
   `(funcall *cont* ,@retvals))

(defmacro =funcall (fn &rest args)
   `(funcall ,fn *cont* ,@args))

(defmacro =apply (fn &rest args)
   `(apply ,fn *cont* ,@args))

(defparameter *paths* nil)

(defconstant failsym '@)

(defmacro choose (&rest choices)
   (if choices
      `(progn
         ,@(mapcar #'(lambda (c)
                        '(push #'(lambda () ,c) *paths*))
             (reverse (cdr choices)))
         ,(car choices))
      '(fail)))
```

```
(=defun non-deterministic-pick (someSet)
  (let* ((poss (pick someSet)))
     (choose-bind x (cdr someSet)
       (if  (equalp x poss)
;;;             (= 1
;;;                (pop-up-message-dialog
;;;                  *lisp-main-window*
;;;                  "demonstrate pick"
;;;                  (concatenate 'string "accept " (format nil "~a"   x))
;;;                  warning-icon "Okay" "Next")
;;;                )
          (=values x)
          (fail)
          ))))

(defmacro choose-bind (var choices &body body)
   `(cb #'(lambda (,var) ,@body) ,choices))

(defun cb( fn choices)
   (if choices
      (progn
        (if (cdr choices)
           (push #'(lambda () (cb fn (cdr choices))) *paths*))
        (funcall fn (car choices)))
      (fail)))

(defun fail()
   (if *paths*
      (funcall (pop *paths*))
      failsym))

(=defun two-numbers ()
  (choose-bind x (from-to 0 100);'(0 1 2 3 4 5)
    (choose-bind y (from-to 0 100);'(0 1 2 3 4 5)
      (=values x y))))


(=defun parlour-trick (s) ;; after Graham
 (=bind (n3 n4) (two-numbers)
  (if (= (+ n3 n4) s)
     `(answer is ,n3 ,n4)
     (fail))))
```

Appendix F (library in Haskell)

```
--reservations :: Book -> [Borrower]
type Book = String
type Borrower = String
type Loans = [(Book, Borrower)]
type Library = ([(Book, [Borrower])] ,Loans)
type Report = String


reserveBook :: Library -> (Book , Borrower) -> (Library, Report)


reserveBook  lib (b, m)
  = let
        (reservations, loans) = lib
        reservers = apply reservations b
    in
        if (m `elem` reservers)
        then
                (lib, "alreadyReservedByThisBorrower")
        else
                let
                   newReservers = reservers ++ [m]
                   newReservations = reservations `override` [(b,newReservers)]
                   newLib = (newReservations, loans)
                in
                   (newLib, "success")

replace maplet [] = error "replace without match"
replace maplet (hd:rest)
     = let
                 (d, x) = maplet
                 (dd, y) = hd
        in
                if (d == dd)
                then
                        (maplet:rest)
                else
                        (hd:(replace maplet rest))



override [] l = l
override l [] = l
override l ((b,bs): rest)
        |elem b (dom l) = override (replace (b,bs) l) rest
        |otherwise = override ((b,bs) : l) rest


dom ::[(a,b)] -> [a]
dom = map fst

apply :: [(Book, [Borrower])] -> Book -> [Borrower]
apply [] _ = []
apply ((b,bl):rest) arg
        |b==arg = bl
        |otherwise = apply rest arg
```