Applying visualisation to model-based formal specifications.

PARRY, Paul William.

Published version

PARRY, Paul William. (2004). Applying visualisation to model-based formal specifications. Doctoral, Sheffield Hallam University (United Kingdom)..

Applying Visualisation to Model-Based Formal Specifications

Paul William Parry, B.Sc. (Hons)

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

January 2004

Abstract

The most important and challenging activity in developing new software systems is arguably ascertaining their features and characteristics before development takes place. This activity, known as requirements engineering, involves software developers identifying the requirements of the customers who are procuring the system, and then documenting them in a requirements specification.

Producing a requirements specification is a complex, time consuming and human-centred activity. It is essential that both parties discuss the requirements, analyse them and negotiate any issues, uncertainties or conflicts that arise. To assist in this process, a prototype of the software can be developed and then thrown away after the requirements process has been completed. Such a prototype helps to stimulate discussion and to provide a vehicle for experimentation and evaluation. This form of prototyping is now a popular and well-known requirements engineering technique.

One powerful throwaway prototyping approach involves developing prototypes quickly using executable model-based formal specifications. These are based upon mathematical notations that possess a defined syntax and semantics. They have a useful dual role in the requirements process. On the one hand, they can be used to express requirements specifications in a precise and unambiguous manner, whilst on the other they can also be subjected to execution to produce a prototype.

However, despite the benefits that such executable specifications have for the developer, their use can be problematic in situations that involve communication with customers. This is because traditionally, for reasons of productivity, the execution behaviour of prototypes developed in this manner is often depicted using developer-centred representations. Such representations often do not correspond to the perceptions or expertise of the customer, as they are often too abstract or technical. If the customer cannot recognise or comprehend these, accurate evaluation of the prototype cannot take place, stifling much needed dialogue and rendering the prototyping process ineffective.

This research advocates that applying visualisation to this form of prototyping can alleviate the problems of comprehension and the subsequent breakdown in dialogue. The objective is to employ the techniques and principles of visualisation to transform the developer-centred prototype execution behaviour into customer-oriented representations based upon pictorial and graphical forms from their own universe of discourse. Applying visualisation in this way can retain the advantages of using executable formal specifications to build prototypes, while at the same time stimulating and sustaining effective dialogue between developers and customers.

The objective of the research concerns the production of a system for visualising the execution of a specific executable formal specification-based prototype development technique. The resulting system is then evaluated by demonstrating its application in a series of case studies. These reveal the capabilities of the approach, and demonstrate the benefits that can be gained over and above the use of existing prototyping techniques based on executable formal specifications.

Keywords: Requirements engineering, prototyping, executable model-based formal specifications, visualisation.

Acknowledgements

A work of such magnitude as this is not undertaken and completed without the help, encouragement and motivation from colleagues, friends and family.

I would first like to express my gratitude to Sheffield Hallam University for providing the opportunity and resources to complete the work. I would also like to extend many thanks to my friends and colleagues within the School of Computing and Management Sciences at Sheffield Hallam University who have offered advice and encouragement throughout the duration of the research.

I would also like to acknowledge the work of Dr. Richard Hibberd, who developed the ZAL system, and Dr. Graham Buckberry who developed TranZit. The work in this thesis integrates closely with these systems.

In addition, I would like to thank my good friend Jo Sutherland for proof reading the thesis after its completion. This, I realise, was no small task.

A great deal of thanks must be expressed to my supervisory team, Dr. Mehmet Ozcan and Mr. Ian Morrey, for their enthusiasm, guidance, and technical expertise. However, this thesis would not have been written without the generous support and encouragement of Professor Jawed Siddiqi. Indeed, I have been extremely fortunate in having Jawed as my Director of Studies. His direction, constructive criticism and help whenever I have needed it have been very much appreciated. I have the utmost respect for him and will always be in his debt. Thankyou is simply too small a word.

Last, and by no means least, I would like to express gratitude to my family. My sister Nicola and Brother-in-law Shaun and their family, and my grandmother Emma, all deserve recognition as they have all offered support during the research. The most gratitude must go to my mother, Margaret. She has supported and encouraged me during all my studies, from school, throughout university and throughout my doctoral research. Her strength, determination and love continue to be a source of inspiration in my life. Finally, these acknowledgements would not be complete without mentioning my late grandfather Bill Rollinson. I just wish he were here now to see how far I have come.

Paul William Parry, B.Sc. (Hons)
January 2004

Table of Contents

Chapter 1 – Introduction

Chapter 2 – Background and Motivation

Chapter 3 – An Alternative Prototype Execution Visualisation System

Chapter 4 – Case Studies

Chapter 5 – Critical Evaluation and Conclusion

Chapter 1
Introduction

Computing pervades nearly every facet of life in modern society. It is acknowledged that no single technology has impacted society as quickly or radically, or has brought such massive changes in the way lives are conducted. We are now at a point in history where computing and some societies are inextricably linked.

Computing can be partitioned into two distinct fields. The first is hardware technology. This comprises the electronics and machinery that provide the underlying processing and communication capabilities. The second field is that of software.

When viewed in simple terms software is an abstraction that enables us to exploit the ever-increasing power, speed and features of hardware devices. However, when viewed from a deeper human-centred perspective software becomes an enabling technology that allows us to enhance and extend our innate natural attributes and skills – it facilitates our creative abilities, enables us to control complicated processes or activities, and allows us to control devices that perform tasks which would otherwise be too dangerous or too complex to perform ourselves.

Since it is so useful and critical, the demand for software is great. Requests for new software increase annually, placing the onus on a global software development industry to implement progressively more innovative and complex software systems.

The high demand for software far outstrips supply, and the ability of the software development industry to supply it. Often software is delivered that does not perform as originally intended, is delivered later than initially expected, or exceeds its budget estimates. This problem, which in 1969 was termed "*the software crisis*" [Naur69], persists today and its effects have an impact on all modern societies and economies [Gibbs94]. Studies conducted by both industry and academia show that 31% of all new software development projects will be cancelled before completion, and that 52% of projects will suffer budgetary overruns [Standish98]. The reasons for such failure stem from the fact that developing software is a time-consuming, thought-intensive, and sometimes experimental activity, which often requires the effort of a large number of software developers.

In response to this seemingly pessimistic situation, and to counter the difficulties, the software development industry strives to develop a solid base of principles and processes that can be applied by software developers. With academic research and experience from the industry, the continuously evolving discipline of **Software Engineering** aims to provide such foundations [Pressman97].

Due to more than 30 years of software engineering research and application, it is now understood that the development of software follows a general 'step-by-step' process that consists of a series of interrelated activities. Firstly, the functions that are required of the new software are identified and documented. Secondly, the software is designed and implemented with a view to meeting these requirements. Thirdly, the software is tested to detect the presence of errors so that they are rectified before the final activities are performed – that of delivery and use. Subsequently, and as part of an on-going process, the software may be modified to meet changes to its requirements or its environment, or address errors not detected in the development stages. This pattern of software development can be characterised in the form of a 'software lifecycle' [Royce70]. A wide variety of lifecycles have been proposed to suit different software applications, eccentricities with development practices, or to accommodate organisational variations [Boehm88, Davis93], but still, it is widely accepted that these activities are essential in all software development projects.

It has emerged that one of the most pressing difficulties with software development, and one that has been identified as contributing greatly to the cause of erroneous or inadequate systems, and to the software crisis in general, is the first stage of the software life-cycle – that of ascertaining and securing the requirements of a new software application [Brooks87]. The umbrella term **Requirements Engineering** (RE) is used to encompass this early lifecycle activity [Royce70, Davis93, Dorfman97, Swebok03]. It is during the 'requirements engineering stage' of the lifecycle, that the stakeholders (i.e. the individuals or groups with a vested interest in the software system, which include software engineers and developers, customers, users, investors, managers, etc.) converge in an attempt to define precisely what is required of the proposed software. The aim is to develop a view of the proposed system that is shared, agreed and understood by all involved [Faulk97]. The result of the requirements engineering stage is a document, or set of documents, known as a **software requirements specification**, that describes the features that the proposed software systems should possess.

The importance of requirements engineering in the software development process, and the challenges faced when undertaking it, means that it is an important area of study. It is with this in mind that requirements engineering is the focus of the research presented in this thesis. The work aims to address a particular problem that can adversely affect the practical application of requirements engineering. The aim of this chapter is to introduce this work. An overview of the problem being addressed is provided, and subsequently, the specific objectives of the research and the scope of the research contributions are defined. Finally, the chapter will describe the structure of the remainder of the thesis.

1.1 Problem Definition and Research Proposal

Effective requirements engineering is dependent upon dialogue between the developer and customer being established and sustained. This is needed to facilitate discussion about the requirements, so that the customers can elucidate their needs, resolve uncertainties, or negotiate conflicting interests that may exist.

To assist in facilitating dialogue during the requirements stage, a notable and now popular technique has been developed. This technique is **prototyping**. It involves developing a mock-up of a proposed software system to illustrate its behaviour [Brook87]. Prototypes provide stakeholders with an opportunity to interact with the software, before it is fully developed, in a tangible way [Gomaa90, Sommerville97]. Importantly, prototypes become an instrument for stimulating dialogue between the developers and customers. By observing the execution behaviour of the prototype, stakeholders are able to see how a system, or part of it, will function when developed. This leads to discussion about the requirements, which in turn can be used to help resolve uncertainties or ambiguities with the requirements information, or uncover new requirements as the prototyping process progresses [Thebaut90].

Experience with prototyping has shown that its effectiveness in assisting the requirements engineering process is entirely dependent upon stakeholders understanding the prototype's execution behaviour. As a prototype is executed, outputs, such as results from calculations, messages and other indicators of execution progress are generated and displayed.

Difficulties occur if stakeholders find these outputs difficult or impossible to comprehend, as then they will be unable to make evaluations and reasoned judgements about the underlying requirements that they represent. This problem arises when the outputs are presented using terminology, concepts or vocabulary with which the stakeholders are unfamiliar. One particular source of such vocabulary is the domain of software engineering. Developers prefer to use software engineering notations and terms to present prototype execution behaviour, as they are familiar with these and they offer the levels of precision and conciseness that developers require. However, whilst being of benefit to the developer, the use of such languages and terminologies when presenting prototype behaviour can adversely affect the ability of non-technical stakeholders to comprehend it [Cooling94], or as Davis, when describing the ability of a "computer-naïve" customer to understand requirements specifications, states, "*...understandability appears to be inversely proportional to the level of complexity and formality.*" [Davis88]. This presentation problem can affect the quality and richness of the dialogue between the developers and customers, and importantly, the reliability and effectiveness of the whole requirements engineering stage.

The problem can occur irrespective of the display mechanisms used to show prototype execution behaviour. Some prototyping approaches rely on text-based display mechanisms. Whilst showing messages or calculation results, these offer little in the way of attempting to meet non-technical stakeholder's needs in terms of their comprehension requirements. Other prototyping approaches base their displays upon more sophisticated representations. These approaches use graphical presentation techniques from the area of visualisation. The aim is to present information in ways that will initiate or promote understanding by drawing upon the potentially expressive nature of graphical representations. In favourable circumstances these offer much greater 'bandwidth' for portraying complex concepts or voluminous information than alternative textual forms [Myers88, Shu89]. Prototyping systems that employ visualisation techniques do so to amplify the comprehensibility of execution behaviour.

Despite the potential of graphical representations and associated visualisation techniques, this research argues that even their application has produced little reward in terms of facilitating customer comprehension of execution behaviour [Ozcan98a]. This stance comes from observations made of existing prototype approaches where visualisation has been employed. The graphical representations used to depict execution in these approaches, are again often based upon developer-oriented software engineering concepts and terminologies.

The utility, flexibility and potential that visualisation has to offer means that it remains a promising technique for presenting prototype execution behaviour. The flexibility and expressive power of graphical representations enable them to present a practically unlimited range of subject material, values, and results from any domain [Cooling94]. This research argues that these characteristics can be exploited by prototyping. The argument is that prototype execution behaviour can be presented using vocabulary and concepts other than those belonging to the domain of the software developer by using imagery that is borrowed directly from the stakeholders' own domain [Parry95]. This will require techniques that enable the use of a rich repertoire of representations, i.e. representations that are based upon photo-quality images or diagrams, and graphical animation, to depict the imagery from the stakeholder's domain. By employing such flexible and expressive display mechanisms, prototype execution behaviour can be portrayed in ways that non-technical stakeholders might be able to comprehend more readily.

It is proposed therefore, that this research should develop an alternative prototype execution visualisation approach that provides customer-oriented display mechanisms. The proposal is that this approach should differ from existing prototyping systems by making available visually rich display facilities. The objective is to develop a prototyping system that will be capable of presenting the results of executing prototypes in more customer-oriented forms.

1.2 Research Contribution

This research contributes to the field of requirements engineering, and more generally to the discipline of software engineering. In addition, by applying the principles and techniques of visualisation, a contribution to the field of visualisation is also made. The contributions are as follows:

1. A literature review that surveys the state of the art in visualising prototype execution. This surveys current work in applying visualisation to prototyping, simulation, and requirements modelling, by presenting a number of existing approaches. An analysis and critique of these is also offered, and deficiencies identified. The review provides a foundation on which the development of an alternative and more powerful visualisation approach can be based.

2. The development of a system to visualise prototype execution. The system applies visualisation to a specific prototyping facility, and aims to do it in such a way that the general problems identified with existing systems are overcome. This particular contribution also includes the development of documentation that describes the system's requirements.
3. Demonstration of the effectiveness of the system. This is achieved by applying the system to a set of case studies.

To summarise, this research provides a unique contribution to the field of requirements engineering by providing a novel approach to visualising prototype execution behaviour.

1.3 Thesis Structure

The thesis is divided into five chapters. The structure of these chapters and summaries of their content are given below.

Chapter 2

This chapter introduces the key areas that are the focus of the research. Firstly, requirements engineering is described. An overview is presented of its principles, products and processes. The description then focuses on a particularly popular and worthwhile requirements engineering technique, namely that of prototyping. Next, the problem that forms the motivation for this research is discussed which surrounds the problem of presenting prototype execution behaviour in a form that is comprehensible to stakeholders. At this point, the discipline of visualisation is described, including its principles and rationale, as a potential solution to the research problem.

After discussing the research problem and the merits of applying visualisation to requirements engineering and prototyping, a survey of related work in the field is presented. Several approaches that employ visualisation to presenting the output from executing prototypes are described. A critique of each approach is also made to identify potential drawbacks and deficiencies. The deficiencies are collated and general trends identified. This leads to a summary of the disadvantages of existing approaches being presented.

Chapter 3

Chapter 3 presents the substantial contribution of the research. It describes the requirements and design for a system that overcomes the research problem stated in *Chapter 2*. The Chapter is divided into three sections. The first presents an overview of the system. In this, the prototyping facility that is the focus for the application of visualisation is described followed by an abstract description of the system that presents its major components and general architecture. This comprises two facets, namely a process that provides step-by-step prescription of the activities required to undertake visualisation, and a software toolset that provides the actual visualisation capability. The second and third sections elaborate on the process and the toolset respectively.

Chapter 4

This chapter demonstrates the application of the system described in *Chapter 3*. This is done through the use of four case studies. Each one exemplifies a certain aspect of the visualisation system by following the development of a visual prototype for a target specification. First the informal requirements specification is given, followed by the enactment of the process. The results are then presented along with a conclusion.

Chapter 5

The fifth and final chapter concerns the evaluation of the work. This involves evaluating the work with respect to the research contributions and the original research objective. In addition, this chapter presents a discussion of relevant issues that pertain to the research, along with an elaboration of improvements and further research directions that would facilitate a continuation of the work.

Chapter 2
Background and Motivation

The aim of this chapter is to describe the background and motivation for this research in detail. It describes a number of areas that are of particular concern and that provide context for the work. Moreover, the specific problem that the work addresses is elaborated in full.

The chapter is structured into four sections. The first section describes the area of requirements engineering, and prototyping in detail, to provide a background for the work. The second section concentrates upon the specific issue of presenting prototype execution behaviour and the problems inherent in this. In so doing, this section frames the work by introducing visualisation as a means to facilitate prototype behaviour visualisation. The third section presents a literature survey that reviews existing approaches in the field of prototype execution visualisation, from which general shortcomings and disadvantages are identified. The fourth and final section in this chapter offers a summary that indicates the specific objectives of the research, namely to develop an alternative prototype execution visualisation system that overcomes the problems identified with the existing approaches.

2.1 Requirements Engineering

Software is developed in response to the needs of organisations, governments and general market forces. To implement software that will satisfy the needs of these 'customers', software developers must first ascertain exactly what is required. The activity known as requirements engineering is conducted in aid of this.

The critical nature of requirements engineering cannot be stressed sufficiently. It forms the foundation of a software development project by providing knowledge about the customer's needs. This knowledge will be used in all major decision-making and design activities throughout the project [Diaz-Gonzales87]. The importance and need for RE is such that it is recognised throughout industry and academia. It is a stated requirement in major software engineering standards, process improvement schemes and good practice guides [Paulk93, Spice03, TickIt01, Swebok03], and is found in most (if not all) software engineering curricula and subject benchmarks used in education world-wide [SEI03, QAA03, ACM03].

Requirements engineering is challenging, however. The practice of RE has long been recognised as being difficult, with many authors within the software engineering domain proclaiming so. For example, Brooks, in 1987 stated "*The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements...*" [Brooks87], and more recently, Potts claimed "*[it] remains one of the most challenging areas for software developers*" [Potts94]. A variety of problems can impede its effectiveness, leading to errors in the resulting descriptions of the proposed software system. These problems include:

- Establishing an understanding of customer needs early in the software development lifecycle [Faulk97].
- Difficulties in exploring customer needs and helping customers identify what they want [McConnell03].
- Distinguishing and recording what the software should do, as opposed to how it should be built [Davis88, Swartout82].
- Problems of scale, and the inherent difficulties of gathering accurate requirements for large systems that typically involve large numbers of users and developers [Faulk97, Thebaut90].
- Communicating requirements to different, but relevant audiences, and in ways they can be understood by these audiences [Stephens93].
- Managing changes in requirements as a software development project progresses [Scharer81].

If errors are manifest in the requirements specification, they will propagate throughout the entire project, crippling it. Moreover, the effects of these errors often reach beyond the bounds of the software's functionality. It is widely observed that the later an error or an omission is detected, the more expensive it will be to repair [Alford77, Boehm81, Davis93]. In addition, software that is flawed or does not perform as the users expect, as the result of an inaccurate or erroneous specification, will be ill received or may not be used at all. Much evidence exists to support these observations [Boehm76, Daly77, Hooper82, Standish98]. The problems are further compounded by the fact that requirements errors are usually the last to be detected [Gomaa90].

Requirements engineering can be viewed from two perspectives, namely *product* and *process*. The product is the document, the software requirements specification, which states the desired functionality and other characteristics of the software. The process is the collection of requirements-oriented activities that, when performed, contribute to the delivery of the requirements specification. These two aspects will be described in detail below.

2.1.1 The Product of Requirements Engineering

The product of the requirements engineering stage is a document, or set of documents, known as a **software requirements specification** (SRS). The SRS has several roles during, and after, a software development project, including:

- Providing a point of reference for the developers and other stakeholders by stating the functionality of the software system. It is important that this functionality should only state 'what' the software should do, and not describe 'how' the software should be implemented, as this may restrict the developer's choice of possible solutions [Scharer81, IEEE98a].
- Acting as a focal point for communicating, circulating, and exchanging ideas and information about the software.
- Forming part of a contractual agreement between the customer and developer [Zave90].
- Serving as the basis for project costs calculation and scheduling [Dorfman97].
- It may be used in the testing and maintenance stages in the context of providing details of what the software should do and what functionality was originally decided.

To fulfil these roles effectively, the SRS should represent a statement of all the characteristics of the software system. These characteristics can be partitioned into three fundamental categories [Davis93, Sommerville97, Kotonya00]:

- The **functional requirements** of the software. This is a list of what the software should do, its proposed behaviour, and how it should deal with inputs and outputs.

- The **constraints** placed upon the system. These define what the software should not do, which includes its bounds, and any environmental quantities (such as laws or standards) that place limitations on the software.

- Any **non-functional requirements**. These define non-behavioural aspects of the software, i.e. attributes of the system as it performs its prescribed activities. For example, non-functional requirements state the system's desired level of efficiency, performance, and maintainability.

Of importance is the means by which requirements specifications are documented. Much work by the software engineering and requirements engineering communities has gone into developing notations with which requirements can be expressed. As such, a wide variety of notations exist, each offering varying features and different levels of formality and abstraction to facilitate the definition of a specification that possesses the attributes described above. They can be partitioned into two major classes: **textual** and **graphical**.

Textual representations are a seemingly obvious choice for expressing requirements. One such notation is that of natural language. This is due to the richness and expressiveness of general vocabulary. However, reliance on natural language specifications is one of the main sources of ambiguity, and hence difficulty [Parnas77, Meyer85]. Natural language is verbose in nature, which enables the incorporation of irrelevant information, and sometime results in unnecessarily long documents. It is also ineffective at decomposing and structuring requirements. The use of mathematically based specification languages has been advocated as an alternative [Parnas77, Hayes86, Jones90]. These languages are based upon mathematical formalisms and concepts that make it possible to concisely and unambiguously express requirements [Vienneau97].

In contrast, diagrammatic approaches employ graphical notations as the basis for representing software requirements, capitalising on the diagram's ability to present relationships, express abstraction, and portray images. As such, a wide range of graphical notations, each with their own defined syntaxes have been developed. Example of such notations include: the Data Flow Diagram (DFD) [DeMarco78, Gane79] used to represent systems in terms of hierarchical decompositions of data stores and flows between them; State Transition Diagrams (STDs) that specify the states that a system can be found in and the possible transitions between these states [Parnas69]; Petri-nets are diagrams that are used to specify process synchrony during the design of time critical applications and are represented as directed graphs augmented with tokens [Petri62, Peterson81]; and a variety of notations (and associated methods) designed for analysis and specification, including the Entity-Relationship Diagram (E-R Diagram) [Chen76], Structured Analysis and Design Technique (SADT) [Marca88, Ross77] and SSADM (Structured Systems Analysis and Design Methodology) [Weaver98]. Finally, the Unified Modelling Language (UML) [Booch99] offers a comprehensive graphical approach to specify designs for systems in object-oriented terms.

2.1.2 The Requirements Engineering Process

In attempting to produce an SRS, it is essential that a defined and systematic approach be taken. Without discipline and diligence, and a defined set of activities to guide the developer, the delivery of an accurate SRS cannot be guaranteed. To this end, a general process of requirements engineering has been established that consists of three fundamental and interrelated activities, namely **elicitation, analysis and definition** and **validation**.

Requirements elicitation is concerned with acquiring and establishing the nature, features and boundaries of a software system. The product of requirements elicitation is a collection of knowledge that may or may not be structured or refined. The activity of elicitation can be thought of as knowledge transfer from a source (a person, document, etc) to the software developer.

Two major problems arise however. The first is that the knowledge to be transferred is not always readily available in a form that the developer finds useful. The second is that it is often difficult for the developer to actually elicit the knowledge from the source, especially if the source is a human expert. For a human, articulating requirements precisely is unusually difficult, as functions and processes are not easily described [Scharer81]. To alleviate these problems, a number of techniques and approaches have been developed. Some of these employ techniques from sociology and psychology. Goguen describes various elicitation methods including interviews, form analysis, brainstorming, observational study, and facilitated meetings [Goguen93].

After elicitation, the developer possesses a potentially large collection of raw knowledge about the proposed system. In this state, the information may not be in a form suitable for subsequent development – there may be inconsistencies, ambiguities and redundancies that could complicate and impede the project. Such undesirable qualities must be reduced or eliminated, and the results formulated into a useful software requirements specification. The activity of requirements analysis and definition serves this purpose. In analysis, the raw requirements details are sorted and refined, and redundancies discarded. In specification, the refined information is structured in the form of an SRS.

The SRS represents the developer's view of the requirements. It is produced from the developer's understanding of the proposed system, which in turn is formulated from the information gathered during the elicitation stage. Consequently, it is essential to judge whether this view matches the customer's original requests and that no errors have been introduced along the way or that no omissions exist. Validation embodies this activity [Fuji97, Lalioti93, Sommerville97].

From the developer's point of view, validation seeks to answer the question "*are we building the right product?*" [Boehm84]. From the customer's point of view, validation can be thought of as an activity that builds confidence into the product. Validation of the SRS entails customers and developers evaluating requirements details, identifying potential errors or omissions, and negotiating and resolving emerging conflicts. These are complex, asynchronous and communication-rich activities, which importantly, require effective dialogue to be established and sustained between the parties concerned. To this end, a variety of requirements validation techniques have been developed [Fujii97]. Each offers varying level of support for establishing and sustaining dialogue.

The most basic approach to requirements validation involves the customer reading the requirements specification. This 'direct' strategy seems the most simple. However, a major problem affects this approach, rendering it difficult and cumbersome, namely, that the SRS is written by developers for developers and often using specialist requirements notations. Therefore, customers may not be able to comprehend the contents of the specification and hence may be unable to judge its accuracy. Furthermore, customers may not be able to conceptualise how a written specification represents their requirements. At best, customers often find a written specification dull to read, and at worst, they may be unable to comprehend its contents at all, especially if it has been written with developers in mind [Gomaa90, Thebaut90]. This is echoed by Dorfman, who observes "*Requirements documents and specifications cannot easily be read by users*." [Dorfman90].

An alternative to this basic approach to validation can be found in 'goal-oriented' requirements engineering techniques. A goal is "*an objective the system under consideration should achieve*" [Zave97]. The goal-oriented approach can be useful in enabling users to express requirements. Customers can often articulate what they want to do with a system, but cannot give detailed breakdowns of the requirements that will enable them to satisfy these needs. Therefore, goals are used as the basis for discussion and investigation. It should be noted however, that goals are not requirements in themselves; they are abstract descriptions of what the system should do, which may require more than one requirement to fulfil [Lamsweerde01]. Goal-oriented requirements engineering has stimulated a number of methods and tools, including the notable KAOS system [Lamsweerde95] and the related GRAIL approach [Bertrand97].

Another validation approach is that of scenarios. Scenarios are descriptions of examples of system usage [Weidenhaupt98]. They provide a means of describing the activities that comprise the software's functionality and the situations in which the system will be used. Scenarios used in this context are known as *descriptive scenarios* [Potts94, Rolland98]. Scenarios can also provide a framework for asking questions about the user's tasks and how a system should facilitate such tasks [Rumbaugh94], i.e. questions concerning "*how is this done?*" or "*what if...?*" and as such provide an opportunity for requirements elicitation and validation by stimulating and facilitating discussion [Hooper82, Potts94, Heymans98, Sutcliffe98]. Scenarios are gaining much popularity within the requirements engineering community. This is evident by the development of a wide range of tool support and methodological assistance [Filipidou98, Rolland98]. However, it should be noted that scenarios are not complete requirements specifications. Scenario descriptions provide details of only instances of system use, whereas a requirements specification describes the system in general terms.

However, these validation activities are often augmented with a technique that has proved highly effective in developing and sustaining dialogue between developers and other stakeholders during the requirements engineering process. This technique is prototyping. Its aim is to enable customers to play a direct and involved role in the requirements process, by enabling them to see and interact with a model of the requirements, and is described next.

2.1.3 Requirements Prototyping

Prototyping is a well-known and popular requirements engineering approach. The activity involves the development of an executable model of the requirements (or part thereof) that the stakeholders of a software development project can use to experiment on and evaluate ideas with. The aim is to enable the stakeholders to assess the accuracy of the requirements gathered so far, as well as to evaluate each other's understanding. [Brooks87, Luqi89, Gomaa90]. To this end, prototyping embodies the activities of requirements elicitation and validation.

The activity of requirements prototyping mirrors its traditional industrial/engineering counterpart, whereby a prototype is developed and used to demonstrate or investigate the intended features of a product or a manufacturing process before full-scale manufacturing takes place. For software, prototyping comprises an 'evaluate then modify' strategy [Davis92] that is repeated until it is agreed that the prototype reflects the functionality required in the proposed product or that sufficient information about the product's features have been discerned.

For our purposes, the objectives and advantages of prototyping are synthesised as follows:

- To stimulate and sustain dialogue between developers and other stakeholders during requirements validation [Gomaa83].
- To encourage stakeholder participation in the process, and at an early stage of the production process [Gomaa90, Carey90].
- To assist in problem analysis by providing an opportunity for uncovering and understanding the details of the proposed system [Hardgrave93].
- To facilitate a non-linear requirements engineering process i.e. accommodate the re-working and correction of problems and errors found in an SRS [Davis93].

Due to its inherent flexibility, it is not surprising that variations of the prototyping process have been developed to accommodate different types of software development projects and the differences in the strength or level of understanding of requirements [Davis92]. One variation is the 'paper prototype' [Thompson92], where an impression of how the software will look and behave is created with paper-based documentation.

Another variant is that of 'evolutionary prototyping'. This approach involves the implementation of a small set of well-understood requirements in order to provide a solid foundation, with additional functionality being appended as the process progresses. During subsequent iterations of this prototyping process, further requirements are revealed or clarified, and correspondingly implemented on top of the existing system [Hekmatpour88, Luqi89, Davis93]. In this approach, the prototype is developed and refined in a quality manner, using quality implementation techniques and tools. Such prototypes embody quality architectural-, performance- and exception handling characteristics.

Lastly, there is the discardable model. This is a prototype that is metaphorically 'thrown away' once sufficient understanding of the requirements has been generated. The experience and knowledge gained during the process contributes to completing the SRS [Gomaa86]. In this approach, the prototype need not be a flawless implementation or offer the same performance as a full-scale production-quality system [Brooks87]. In addition, such a prototype need not model all the requirements of the proposed system. It need only focus upon issues that are believed to require exploration or further clarification. These prototypes can be constructed in a variety of ways, including the use of conventional programming languages, fourth generation database tools, or executable formal specification languages. In addition, specialist prototyping environments or workbenches that facilitate rapid application development can be used [Hekmatpour88].

Although the flexibility and the non-linearity of the prototyping process are useful attributes that make prototyping appealing, arguably its most important characteristic is its ability to facilitate and sustain dialogue between developers and other stakeholders [Gomaa83, Hardgrave93]. Promoting this interplay between the involved parties in a software system's development is critical. From the non-development stakeholder's perspective, they must be given a continuous opportunity to articulate their needs during the early states of a software development project. They must be consulted on decisions regarding requirements, and be instrumental in eliminating developer's misconceptions or clarifying assumptions. From the developers perspective, they must be given the continuous opportunity to clarify and enhance their understanding of the requirements, and resolve ambiguities, contradictions, or fuzziness in the requirements details. Both parties must be given the opportunity to evaluate and review requirements details as they emerge, in order to stem errors that could propagate into the subsequent development. These activities require a 'communication rich' environment where dialogue is continuous. Such an environment can be stimulated and sustained by employing prototyping [Carey90].

Prototyping facilitates and sustains dialogue by regarding the prototype as the focal point of any discussion. Interaction with, and evaluation of the prototype stimulates discussion about its features. Customers can judge, assess, and then discuss its functional accuracy, and subsequently contribute additional materials or provide clarification where necessary. Moreover, developers can use it as the basis to discuss the requirements the prototype represents and use it as a framework to guide them through the discussions about any fuzziness in their understanding. Both parties can evaluate the prototype against their own perceptions of the requirements. They can also evaluate the perceptions of the other party and gauge their reactions during the discussions. These interactions provide customers and developers with the opportunity to express approval or dissatisfaction with an aspect of the prototype, or discuss potential omissions.

As a result of the interaction and dialogue, developers become more aware of the customer's needs. They can validate their newfound knowledge by modifying the prototype to reflect their understanding of the requirements. This process iterates until both parties are confident that sufficient detail about the requirements has been established. This cycle, of evaluate then modify, enables the requirements knowledge to be refined with respect to time. In some forms of prototyping (such as 'throwaway' forms), this time may be quite short, with modifications being made to the prototype almost continuously, whereas in others (i.e. 'evolutionary' forms), the modifications occur over a longer timeframe and in a quality manner, whereby the prototype is grown into a more mature product.

After the prototyping process has been completed, and depending on the form of prototyping being used, either full-scale development continues using quality methods and techniques, as is the case with throwaway variants or the prototype is used by the customers in the same way as a full scale production-quality system would be, until more functionality is added.

2.2 Presenting Prototype Execution Behaviour

The opportunities for iteration and communication offered through prototyping have made it a useful technique that has been applied, in its various forms, to a great number of software development projects. Its popularity and adoption within industry, and its study within academia, illustrates its usefulness and importance. However, a variety of issues and problems can effect the successful application of prototyping. These problems include problems of a technical and managerial nature, such as determining when to conclude the prototyping process, how to control and update the documentation that the process can generate, and how the stakeholders can be reassured that the prototype is not the full working system.

Of particular importance is a problem that can adversely affect what is arguably prototyping's greatest strength – its potential to stimulate, sustain and enrich the communication between stakeholders about the requirements of a proposed software system. This problem is that of presenting the prototype's execution behaviour to stakeholders effectively. Prototype behaviour is generally presented as the results, messages, or inputs/outputs that are generated from the prototype's execution. If for any reason, these results are not presented in a manner that the stakeholders can understand during evaluation of the prototype, validation of the requirements that the prototype represents then becomes uncertain and unreliable. This is because stakeholders will not be able to make a reasoned assessment of it [Parry95, Ozcan98a].

This section addresses this presentation problem. First, a series of critical factors that contribute to the problem are examined. Next, a potential solution, in the form of applying visualisation to present prototype execution behaviour, is elaborated. Finally, challenges in applying visualisation to prototype execution are described.

2.2.1 Critical Factors in Presenting Prototype Execution Behaviour

The critical problem of presenting prototype execution behaviour in a way that is comprehensible to stakeholders during the validation process is symptomatic of a combination of several interrelated issues, namely *support*, *presentation*, *developer attitudes*, *development costs*, and the *inherent differences between developers and other stakeholders*.

Support concerns the presentation facilities that are available in the tools, languages and workbenches that facilitate prototype implementation and execution. It is evident that many of these simply do not offer comprehensive support for presenting prototype execution, especially those that facilitate the development and exploitation of throwaway prototype variants. Many prototype implementation approaches shun comprehensive presentation techniques in favour of providing focused and productive prototype development environments. In addition, these approaches usually accompany specialist notations that are used to express prototype behaviour. Such notations are often abstract, in that they obviate the need to specify behaviour in low-level terms. This abstraction also displaces comprehensive display features in favour of general, and often minimal, display and user-interface facilities. While being of benefit to the developer, these abstract or minimal presentation techniques limit the range of representations with which prototype behaviour can be presented. The types of representations usually range from minimal text based outputs to rudimentary diagrams. However, in many cases the only representations offered are based upon the concepts, terminologies and vocabularies used by developers, i.e. those belonging to the domain of software engineering. It is unreasonable to expect customers to learn and understand such regimes, which in turn, leads to the situation where customers do not comprehend the prototype behaviours that are presented to them [Lalioti93]. This factor is directly related to next factor, *presentation*.

Presentation factors concern the nature of the representation used to depict execution behaviour. When displaying the progress of execution, the various results, messages or input/outputs are presented to the stakeholders. For this presentation to be effective, the types and styles of the representations, from which the displays of results, messages or input/outputs are constructed, must be sufficiently 'rich', i.e. be based on a wide range of flexible and comprehensive appearance types. Such representations are required so they are able to accommodate a variety of stakeholder types and domain characteristics, and deliver displays that can be tailored to suit their needs so they can comprehend the underlying requirements successfully. If however, prototyping approaches do not offer suitably rich and flexible display mechanisms the representations used will not portray the prototype's execution in ways that suit the stakeholders and hence, in ways they will understand. This can seriously impede the overall effectiveness of the prototyping approach.

In addition, effective presentation is not just dependent upon varied and flexible representation types. It also relies upon showing execution behaviour in meaningful and unambiguous terms so it is not misinterpreted or misunderstood by stakeholders [Parry00]. In other words, representations must adhere to the concept of 'what you see is what it means' (WYSIWIM).

Developer attitudes concern the preferences of developers when producing and using prototypes. Developers often prefer to use technically oriented terms with which to present prototype execution behaviour. They might not wish to use different, less technical or less precise notations during the process. Instead, they are likely to be very familiar with such representations as they may well have extensive practical experience in using them. This has a direct influence on the types and styles of the representation chosen to depict prototype behaviour.

Development costs concern the financial cost, time and effort required to develop prototypes with comprehensive presentation characteristics. To reduce cost and effort, a developer may chose not to implement fully functional customer-oriented user interfaces that employ rich presentation techniques. User interfaces can consume much development effort - it has been suggested that they can take between 30% to 80% of the effort of developing a software system [Myers92, Remington97]. This particular point is pertinent to throwaway prototyping applications where the investment in terms of time and costs may never be recovered.

The *difference between developer and other stakeholders* concern the difference in technical knowledge between developers and customers [Ozcan98a]. It is important to recognise that fundamental differences do exist, and are manifested as divergences in terms of the knowledge of each other's respective domain. On the one hand, the developer understands the domain of software engineering and its associated languages, terminologies and concepts, and on the other, stakeholders understand their own domain, their working environment, and the activities that are associated with their domain. Additionally, when discussing or describing their domain, each group often prefers to use their own terminologies, vocabulary, and shortcuts. Due to these differences, a 'comprehension gap' may emerge between the parties involved. This creates an invisible barrier between them. Whilst not immediately apparent, such differences will produce cultural and technical divides that will affect communication. Customers may feel ignored due to a lack of participation in the process, feel they may be unable (and subsequently unwilling) to articulate their needs to the developer, or feel that the developer does not (or is unwilling) to understand them and see their point of view. Developers may feel insecure about the software they are developing as they may lack confidence in the requirements that are emerging, or they may have little faith in the customer's ability to communicate their desires. These points can affect developer morale and the eventual success of the project.

Although proponents of prototyping claim that it is possible to overcome the problems that stem from these factors, in practice it is found that they still manifest themselves in the prototyping approaches in use today. This renders many ineffective at depicting prototype execution behaviour in ways that all stakeholders can recognise and comprehend.

However, it is argued by many prototyping practitioners that a solution to this problem can be found by borrowing certain principles from an area that has, from the outset, been developed to address presentation problems. This area is **visualisation**. At its heart lie techniques that can be utilised to enhance and amplify comprehension of information by presenting it using graphics. Such techniques have been applied to prototyping, whereby the results of prototype execution are presented using diagrams and pictures to improve understandability and promote stakeholder comprehension. This notable approach has been applied widely in a variety of prototyping situations. It is described in detail in the following section.

2.2.2 Applying Visualisation to Present Prototype Execution Behaviour

As a way of overcoming some of the difficulties that are inherent with presenting prototype execution behaviour, the application of the technique of visualisation has been advocated [Cooling94, Parry95]. Visualisation is an activity that involves presenting information using visual representations, i.e. representations that utilise the power of computer graphics. The aim is to exploit the power of visual representations to amplify the comprehensibility of information. To this end, visualisation can be viewed as a tool that can be used in situations where the inherent expressive power of visual representation might be leveraged against presentation problems.

The rationale for applying visualisation to prototyping rests upon its potential to assist in stimulating communication between the customer and developer. Through visualisation, visual images, based on visually rich and flexible displays techniques, can be used to portray execution information in terms borrowed directly from the customer's own domain with which they are familiar. If the customer is able to comprehend the execution behaviour through such appropriate images, then they will be able to make reasoned comparisons between their own perceptions and what they see, or identify omissions in what is presented to them. Subsequently, they will be able to discuss the differences or omissions that they recognise with the developer. The communication in this visualisation-assisted prototyping process is likely to be effective and productive.

Visualisation is beneficial to both the developer and customer. From the developer's perspective, confidence in the project can be built on the knowledge that the requirements are an accurate reflection of customer's needs. Confidence breeds motivation, and in turn, results in higher probability that the project will be a success. Such benefits accrue from a successful requirements engineering process, which can occur if supported by effective communication through visualisation. Customers can feel confident and content that they have played a role in the development process. They see progress, and they see specifications that more accurately document their needs.

Visualisation stems from the development, throughout the 1980s, of computer graphics. Rapid technological advancements and reduced costs facilitated the development of a highly flexible output medium. Corresponding advancements in software took advantage of the high-resolutions and colour capabilities of this medium, displacing older text-based displays as the primary form of computer output. This is evident by powerful high-resolution user-interfaces and use of photo-realistic images that pervade every facet of computer use today.

Mass adoption of such technology has enabled graphical presentation techniques to be harnessed and exploited to portray the results that emerge from applications. These efforts are driven by knowledge that presenting information graphically, in many circumstances, offers a greater degree of flexibility, abstraction and expression, than is offered by text-based displays. These characteristics, coupled with advancements in technology, and the need to alleviate problems with existing output methods, have led to development of visualisation [Frenkel88, Spence01].

The origins of visualisation stem from the need to display increasingly complex and voluminous information. Traditional display methods, as used through the 1960s and 1970s, were limited in their scope, with output often restricted to rudimentary one-dimensional text streams. Visualisation provides the techniques necessary to alleviate these presentation problems by presenting information visually.

Visualisation is not merely an alternative display technique, however. When applied effectively it allows information to be portrayed in ways to directly promote or amplify comprehension. It enables values to not just be 'seen', but instead, it allows insights to be gained, perhaps by enabling relationships between data to be discerned, by enabling patterns from complex arrangements of information to be extracted, or by presenting information in forms that accommodate the viewer's 'visual requirements' [Gershon98].

Much has been documented about the superiority of graphical representations over textual ones in certain circumstances. Many authors make claims or discuss observations about the characteristics of presenting information visually and how this facilitates and promotes comprehension. For example, Stasko claims, "*the two-dimensional format of a picture can provide greater amounts of relevant information more fluently than a stream of text*" [Stasko92], and Barrett states "*pictures leave a more lasting impression that words alone*" [Barrett94]. With another similar claim, Nan Shu argues that "*pictures can help understanding and remembering*" [Shu89].

In addition to the general claims outlined above, many authors state the impact of the characteristics of visual representations in certain contexts or applications. One such application is that of representing the complexity inherent in information [Spence01]. Certain types of visual representations, e.g. diagrams and charts, can be effective at expressing situations that would otherwise challenge the abilities of textually-based offerings. For example, Larkin [Larkin87] provides a good example of the contrast between visual and textual representations by presenting a problem concerning the arrangement of a pulley assembly. It is obvious from the representation that Larkin uses that the arrangement can be easily and instantly discerned, whereas if a textually based representation is used, the complexity exceeds the level by which it is possible to instantly and accurately imagine the assembly. In addition, Cox claims "*In general, pictures provide a better representation for most complex structures....*" [Cox89]. This may be attributable to the size of the mind's working memory, or the ability of an individual to understand a scene and conceptually derive a mental image of it.

A particular advantage of visual representations is that they may provide a way of alleviating certain communication problems [Camara94]. For example, pictures can be used to express thoughts without the use of text/language and overcome some cultural differences, or as Nan Shu states, "*When properly designed, pictures can be understood by people regardless of what language they speak.*" [Shu89]. This is evidenced by the proliferation of internationally recognised symbols and icons, such as road signs or the 'Hazchem' scheme used to indicate hazardous chemicals [TSO01].

These characteristics have led to visualisation proliferating in areas where viewer comprehension is of paramount importance. Such areas range from science to medicine, economics and business [Mantey94]. For example, within the domain of physics, mathematical models are portrayed graphically to facilitate greater insight into them [Sprott97, Folin92]. In the field of medicine, data obtained from Magnetic Resonance Imaging scanners can be co-ordinated and presented as three-dimensional images from which medical personnel can make accurate and informed judgements as to the medical condition of patients. In the domain of business and economics, economic models for forecasting money markets or for analysing financial trends can be depicted as visual representations, as well as business processes themselves [Barrett94]. For examples of such systems, refer to [Hagen00], the proceedings of the IEEE's series of annual visualisation conferences [Visualisation02] and the ACM's Special Interest Group on Graphics and Interactive Techniques (SIGGRAPH) conference series [SIGGRAPH03].

While presenting voluminous data in a visual form is an effective application for visualisation, the same techniques can be applied in different ways to support the visualisation of other types of information. Visualisation is often used to facilitate the direct display of representations. This type of application includes the visualisation of software (i.e. source code, and object code that is under execution) to render it in a more comprehensible form for developers [Myers88, Green91, Reiss01]. This is especially useful for the development and maintenance of concurrent software [Pillet95]. Other aspects of software and its development process can also be visualised, for example software architectures (i.e. module interdependencies, and object-oriented relationships) [Carr95, Feijs98, Booch99]. Computer-based applications can themselves be visualised, for example, visualising the contents of databases for the purpose of information retrieval [Combs92, Walton94]. The topology of communication networks and the volume and direction of network traffic also lend themselves to visualisation [Martin93]. Lastly, visualisation has been applied to illustrating the structure of the World Wide Web and the relationships that exists between Hypertext documents [Benford97].

Visualisation has also been extended to cover other domains. For example, visualisation has been used to depict the use and structure of natural language [Narayanan95]. It is also claimed that visualisation can play a major role in teaching, whereby complex concepts can be presented as diagrams to pupils and students [Dobson94].

The nature of visualisation is rooted in the relationship between information and representation. For the purpose of this research, information is viewed as consisting of two related aspects: the inherent content or meaning that the information possesses; and a representation that renders it accessible to a viewer. The knowledge provides 'value' whereas the representation provides an interface between the value and a viewer/reader. The representation is fashioned as an outward face or appearance. Visualisation is essentially a transformation process that transforms this appearance into alternative forms. The alternative forms being visual, and with the intention of being more appropriate to the viewer's needs. However, it is desirable for the transformation process to be performed in a way that does not change the value of the underlying information – visualisation may alter the viewer's perception of that information, but it should not alter the information itself.

In traditional visualisation systems, such as those that facilitate scientific, mathematical or volume visualisation, the transformation processes that underpin them are encapsulated in a set of rules that state how the source representation will be re-presented. The rules would state how to process and analyse the source representation or source information, which visual representations to apply, and how to render these on a chosen display.

In contrast to the technical details of how visualisation is performed, an important issue, and one that has profound significance to visualisation, is interpretation. Interpretation is a pertinent issue in many domains, such as art, law, and philosophy. It is necessary therefore to define its scope and meaning in the context of visualisation, and more specifically to this research, to its relevance.

In terms of visualisation, interpretation, or more specifically, the act of interpreting a visual representation, refers to a transaction between content and a viewer. This transaction, concerning the determination of meaning from the source, is conducted through a visual representation that portrays the content in a way that its designer/creator deemed appropriate. The transaction is thought of as being successful if the viewer accurately determines the knowledge as intended, from its representation.

A successful transaction depends upon two conditions. The first is that the representation must be appropriate at portraying the knowledge, i.e. its appearance is suggestive, indicative or directly corresponds to the knowledge concerned. In some cases, to accommodate a particular viewer type, the representation must be composed of visual cues with which a viewer is familiar. The second is that the viewer must be able to understand the representation sufficiently to ascertain the knowledge it attempts to convey. The viewer may require a certain level of knowledge or experience to understand a representation. For example, the viewer may have to understand a vocabulary of symbols, or be capable of reading a particular diagram or chart. If these conditions are not satisfied, the representation may be misinterpreted, resulting in the viewer determining an incorrect, or a partially correct perception of the knowledge.

From this position, an explanation of the objective of the visualisation process can be derived. Visualisation attempts to deliver a representation of the knowledge that enables the transaction (i.e. knowledge transfer) to be successful. Its objective is to transform a given representation into one based upon graphical forms, that is able to convey the underlying knowledge in a manner that reduces the potential for misinterpretation and promotes viewer understanding.

2.2.3 Challenges in Visualising Prototype Execution Behaviour

A variety of fundamental challenges confront the application of visualisation to presenting the behaviour of prototype execution. These challenges can be divided into two sets. The first involves *human factors* challenges that pertain to cognition and comprehension of visualisations, and the second involves the *technical and practical* aspects of visualising prototype execution behaviour.

Human factors pose fundamental and profound challenges. The first of these involves understanding how visualisations are indeed useful and how appropriate information can be imparted, and subsequently comprehended by a viewer. Understanding this can form a way to assist in building effective visualisations. Humans possess a powerful image acquisition and processing system, the anatomy of which is well known. However, problems begin to arise when a deeper understanding of what happens to an image after it has been captured is required, as the precise reasons for the way meaning can be grasped by 'seeing' a visual representation are not entirely understood. Some of the more general ideas that have been postulated by various authors about image comprehension will be highlighted.

Central to comprehension is the question that relates to how information can be extracted from the collection of shapes, colours, and patterns that make up a visual representation. When considering this issue, the constitution of the visual representation must be taken into account. A visual representation consists of a certain number of visual cues, or elements, by which information can be expressed [Domik93]. Additionally, the relationship between visual cues and their contribution to the overall content of a complete visual representation is important [Lohse94]. One particular belief is that each visual cue plays a role in conveying information, or a piece of it; for example Domik suggests that "*a picture is the sum of its visual cu*es" and that pictures need combinations of various visual cues to be effective [Domik93]. This belief comprises the 'component-view' of comprehension, whereby it is argued that a representation is analysed by the human brain and understood by decomposing it into its constituent parts. Each part is processed, with meaning attributed to each part (based upon comparisons to existing patterns or images in memory) in parallel, resulting in a real time mechanism that facilitates comprehension.

In contrast to the component-view, there is the overall or 'synthesised-view' of a how an image is understood, or in other words, how the whole representation conveys meaning. It is argued that to understand a text-based representation, a reader must focus upon each word and understand it [Petre90]. There is a temporal issue here, which dictates that to understand a sentence, the entire set of words must be read in their correct sequence, whereby each word's meaning must be retained then pieced together. Visual representations on the other hand do not suffer from this, as they can be 'scanned' quickly, and the whole image can be 'absorbed' [Petre90]. Converting large data sets into visually represented models therefore greatly assists in its rapid comprehension [Webster90]. However, to gain an overall understanding of a large set of data, if viewed by textual means, each data item would need to be read and understood.

The next human factors challenge progresses to understanding how the visual processing system in humans is able to understand the image characteristics. Humans excel at acquiring and processing visual imagery. They achieve this by invoking the highly developed visual processing system that comprises the eye, the optic nerves, and the visual cortex [Valeric03]. Anatomically, these areas are well known, but modern medical science cannot fully explain how these bodily components combine to provide the image capture and recognition system inherent in humans. Despite this fact, many authors have stated claims about the observable capabilities of the human visual processing system. For example, many authors have commented on the perceptual endowments of people and that they are strongly optimised for real-time image processing [Duisberg87, Myers88]. Other authors comment on the 'bandwidth' of the eye-brain combination and how it is the most powerful human communication channel, for example [Webster90], and that it is especially efficient when applied to analysing pictorial information. For a more in-depth treatment of the visual abilities of humans, refer to [Sekuler94, Humphreys89].

The final human factors challenge is in understanding how humans perceive visual images. When the eye has received visual information, it travels along the optic nerves into the brain. The images are passed to an area of brain known as the visual cortex which is connected to the higher brain centres responsible for memory and consciousness [Hung02]. However, the reasons behind the ability of humans to understand, recognise, and interpret such visual information is not understood, although many theories have been proposed. Many centre upon the development of 'mental-models' of the world, and how certain visual images correspond more closely to these models [Cox89, Spence01]. However, such imagery also affects how mental models are interpreted. It is also claimed "*the way a problem is represented has a strong influence on whether we can understand and solve it*" [Bocker86]. Indeed, many great scientists have stated that they do not "*think in words*" as observed by Larkin [Larkin87] and that many problem solvers prefer a diagrammatic approach. This lends more credence to the notion of mental models, how mental models are developed, and how comprehension may stem from such models. Another aspect of image comprehension is the process that is invoked within the brain. The perception of spatial relationships and the discovery of patterns activate a series of mental processes. Evidence indicates that these are different to the mental processes invoked when a viewer reads text-based representations [Domik93]. This suggests that visual and textual comprehension is handled by different brain areas and are processed in different ways. For a more detailed investigation into the cognitive aspects of understanding visual representations refer to [Marr82, Spoehr82].

The technical challenges in providing visualisation facilities are as follows. Some of the major concerns that accompany each challenge are also discussed to provide additional insight into how the challenge may be satisfied, or what is required to overcome the problems that the challenge presents.

The first technical challenge is tool support. This concerns the provision of visualisation facilities through software tools and additional supporting systems. Satisfying this successfully hinges on the ability to deliver visualisation facilities to both new and existing prototyping environments and tools. When considering the development of new prototyping support systems, their designers incorporate visualisation features from the initial stages of the tool's development. However, it is the case that a wide variety of tools already exist to support validation through prototyping (for examples, see *Section 2.3*) Also, software developers often have a propensity to repeatedly use the same tools over sustained periods because they are tried and tested, familiar and the investments made in cost/training will not be wasted. To fully exploit the use of these existing tools or retain their usefulness in the future, it would be necessary to augment them with visualisation technologies, possibly via the use of a separate visualisation system. In these cases, it would be imperative for the visualisation system to inter-operate with the prototyping environment, so visualisations could be applied to the execution results.

The second technical challenge is to embrace presentation technologies and techniques that will both enhance the depiction of outputs from prototype execution and offer greater support to the visualisation process in general. Moreover, new technological developments must be monitored with a view to exploiting these for the benefit of visualisation. To satisfy this particular challenge, visualisation tools will be required to possess certain characteristics such as *diversity* or *visual power* of the visual representation and *facilities for composing and modifying representations*.

In terms of diversity, which may also be termed 'visual power', a spectrum of individual visual cues and representation types should be made available from which visualisations can be developed. However, there is a need to express how combinations of visual cues can be used to formulate complete visualisations. In other words, an abstraction that provides a frame of reference with which to discuss or implement visualisations is required. Such an abstraction can be based on a 'visualisation hierarchy' that consists of three levels.

At the first and lowest level in the hierarchy, is the need to provide a range of individual visual cues, supported by graphical animation facilities. These provide the visual building blocks from which any type of visual representation can be developed, and comprise geometric shapes, text elements, and 'larger' cues such as atomic images or charts. The visual cue types should span the range between direct and abstract to provide flexibility to portray prototype execution in ways that are most appropriate given the viewers and the subject material. Direct representations provide realistic presentations of an object or system that is close to its real-world counterpart. Examples of such representations include photo-realistic images. Abstract representations might be used to provide overviews, and examples include diagrams or charts.

At the intermediate level is the need to combine individual visual cues and animations to develop 'visualisation objects' that can be applied to the products of prototype execution. Such execution products might consist of individual results, messages, or any other indicators of execution progress. It is necessary to visualise these at run-time. Examples of visualisation objects might include complete charts, with labels and axes, complex combinations of geometric shapes that form complete images, or images combined with results generated through execution. Also at this level, there exists the need to depict the execution progress in a wider contextual setting. This should involve augmenting the execution progress visualisations with visual objects that reflect how the system could be placed in a real-world context by presenting entities that are external to the system under consideration, or depict relationships between the visualisations of execution progress. It is important to visualise such details to provide a 'setting' into which the visualisations of execution progress can be placed. This is required to provide meaning – data and values are often meaningless when presented by themselves, but instead require a context for them to be understood.

At the highest level in the visualisation hierarchy is the need to present a complete 'scene' that contains combinations of visualisation objects, visualisations for both execution progress and contextual details. Scenes may visualise the execution of significant events. These may include the execution of a part of the prototype, or the complete execution of a particular system function. Importantly, scenes can be likened to frames in a storyboard, where each frame depicts an important event that is part of a larger sequence. When augmented with contextual visualisations, this method enables execution of schemas to be portrayed in a storyboard fashion [Andriole87]. This has advantages, since the basic concept of storyboarding is readily familiar to many, and that storyboards themselves offer a convenient means of visualising sequences of action [Friaioli00, Hart99].

With regard to composing and modifying visual representations, any tool that supports visualisation should provide suitable editing facilities. Moreover, as a consequence of the validation process, the prototype may undergo significant change. Consequently, the visual representations associated with the prototype could also change. Considering the number of iterations during user validation and the importance of timeliness of the validation activity, it is important that a visualisation should provide speedy creation and modification facilities.

The final technical challenge is concerned with overcoming the problem of the *integrity* of visual representations. These problems arise when unsuitable visual representations are used to depict prototype execution. This challenge may also be termed the *correspondence problem*. It stems from the semantic distance that exists between the meaning of executing a prototype and the representations that are used to portray that meaning. Again, this relates to the communication problems attributed to traditional (i.e. non-visual) prototyping (see *Section 2.1.3*). However, in systems for visualising prototype execution, this problem can be more acute because of the nature of the visualisation process – achieving correspondence between execution behaviour and its representation is fundamentally difficult. There are no guarantees that a representation will be relevant, meaningful or unambiguous when applied to depicting prototype execution behaviour, especially if humans play a large part in the creation process. If arbitrary or potentially ambiguous representations are used, then the intended meaning of the behaviour of the prototype may be not communicated to a customer effectively. This will result in misunderstandings arising that have the potential to impede the prototyping process [Parry00]. If a system for visualising prototype execution automatically generates visualisations, using rule-based or algorithmic means and depends upon defined inputs and outputs, this problem can be alleviated to a degree. However, such systems may be considered rigid in that they might not offer the rich, flexible and varied visual representations required by non-developers. Clearly such systems are counterproductive. The correspondence problem is not merely a technical problem, but a wider communications problem where it is necessary to match visual representations to the expectations of the customer.

Our investigations into the challenges presented by human factors and technical issues have elicited useful characteristics of visualisations and of the human visual processing system that can be exploited when applying visualisation to prototype execution.

Following the presentation of the basic principles of visualisation and the rationale and issues that surround its application to presenting the execution of prototype behaviour, the following section presents a survey of existing work in this domain. The aim is to highlight useful techniques and good practice, as well as revealing deficiencies that exist with the current technology. The objective is to provide motivation to develop an alternative visualisation system that can overcome these difficulties.

2.3 Survey of Existing Approaches

This section surveys the current state of the art with respect to existing approaches to visualisation prototype execution behaviour. The aim is to identify shortcomings with the existing approaches that may affect, constrain or limit their utility. The results of this survey are then used as the basis for satisfying the research objective, which is to develop an alternative visualisation system that provides enhanced visualisation capabilities.

To facilitate the evaluation of these approaches, their characteristics are identified and categorised in terms of a taxonomy comprising of a set of criteria based upon the investigation into the principles of visualisation and the challenges presented by human factors and technical issues, as described in *Section 2.2*. To quantify the extent to which a system provides a particular feature, a simple categorisation scheme is offered based upon a series of 'ticks' – no ticks indicate that the system does not provide the given feature, whereas three ticks indicates that the feature is provided comprehensively. A summary of the taxonomy is given in *Table 2.1*.

Criteria	Description	Extent
Application Domain	The scope of the prototyping and visualisation system.	(not applicable)
Visual cues (the lowest level in the visualisation hierarchy)	The individual visual cues available to construct visualisation objects.	The range and comprehensiveness of the visual cues provided.
Visualisation objects (the intermediate level in the visualisation hierarchy)	The basic units of visualisation available to portray the execution of prototypes.	The range and diversity of the visualisation objects provided.
Scenes (the highest level in the visualisation hierarchy)	Compositions of visualisation objects that indicate overall views of the prototype behaviour.	The extent to which the system permits scene level visualisations to be created and used..
Dynamism	The way in which dynamism is presented	The facilities available for dynamic visualisation.
Flexibility	The mechanisms available for composing and modifying visualisations.	The extent of the facilities available for composing and modifying visualisations.
Representing Relationships	Mechanisms available to facilitate the depiction of relationships between visualisation objects.	The amount of facilities available for representing relationships between entities in the given visualisation.
Abstraction	Any abstractions mechanisms available to enable overviews of execution to be generated or to enable viewers to focus on individual aspects of it.	The amount of mechanisms provided to portray abstraction.
Correspondence	The level of correspondence/integrity between the visual representations and the underlying meaning of the execution of the prototype.	The extent of between specification/model and the visualisation.

Table 2.1*. The criteria used to classify and evaluate systems to visualising prototype execution behaviour.*

2.3.1 Teamwork/ES [Blumofe88]

Although many CASE tools exist that allow SA/RT (Structured Analysis with Real Time extensions) [Ward85] diagrams to be drawn neatly, they are limited in so far as they do not allow the resulting diagrams, or models, to be executed. The *Teamworks/ES* system has been designed to achieve this, i.e. execute real-time structured analysis specifications in an interactive and graphical manner for promoting understanding of the system being modelled.

The system allows SA/RT diagrams to be drawn and modified by means of a direct-manipulation style editor and then initialised with 'tokens'. Tokens, which are represented on the diagrams as black squares, can be placed in various locations, but their placement must conform to the semantics of the diagrams. Firstly, tokens can be placed on flow lines to show activity on that flow; and secondly, they can be placed on bubbles (or processes) to represent that the bubble is capable of transforming its inputs into outputs (i.e. the bubble or process is *enabled*). The SA/RT notation also allows abstraction by further decomposing processing elements into lower-level DFDs.

Since real-time systems are the focus of *Teamworks/ES*, a number of features for specifying and analysing the timing requirements of the models are provided. Fundamentally, when a model is under execution, *Teamworks/ES* supplies a global 'clock' that defaults to 1 Hz, and all operations and events are synchronised with this. It is possible to specify delays for certain processes to simulate the notion of real-time execution. This feature can also be disabled, thus giving the impression that each process completes instantly and consumes zero time. Execution of lower-level DFDs can be controlled. They can execute normally, i.e. in real-time, or they can be made 'transparent' in that they simply return their results instantly, thus helping to speed up model execution. The overall speed of execution can be controlled, with options from full-speed to slow, and a pause feature is also provided. The results of the execution are reflected graphically on the SA/RT diagrams, with tokens being placed and removed according to changes in the state of the model.

Conclusion and Critique

The *Teamworks/ES* system is tied to a particular diagrammatical modelling approach, i.e. SA/RT diagrams, and hence the range of visualisations offered can be somewhat limited. The visualisations are often based upon the notation and formalisms used in the underlying approach (which is in fact the software engineering/information systems domain), This could cause the customer to experience comprehension difficulties.

Criteria	Description	Extent
Application Domain	Real-time systems	
Visual cues	Simple geometric shapes combined to form SA/RT primitives	✓
Visualisation objects	Graphical elements corresponding to SA/RT diagram primitives.	✓
Scenes	Complete SA/RT diagrams.	✓✓
Dynamism	State changes in the executing model are reflected in the resulting visualisation as changes to the graphical primitives.	✓
Flexibility	Visualisations easy to modify via manipulation of SA/RT diagrams through supporting software.	✓
Representing Relationships	Possible to state position of entities within the SA/RT diagram and connect them to others via arcs.	✓✓
Abstraction	Supports hierarchical decomposition of SA/RT diagrams.	✓✓✓
Correspondence	A degree of correspondence between script and resulting visualisations. Problem surrounds the production of the underlying scripts, in that it is impossible to determine if these meaningfully depict the requirements.	✓✓

Table 3.1. *Classification of Teamworks/ES.*

2.3.2 Mosel-MetaFrame [Margaria98]

This system attempts to combine formal methods with a visualisation system to enhance and improve user- and developer-accessibility during validation and verification of hardware circuits. The aim is to make the formal specification technique used to describe the operation of the circuits more accessible.

The characteristics of the approach are: i) it enables models of hardware circuits to be expressed using a formal specification language, specifically monadic 2nd order logic. This allows the operation and properties of the circuits to be defined, which then enables the circuits to be subjected to mathematical treatments to detect errors; ii) it provides a visual programming environment by which visual models can be constructed and associated with the formal specification; iii) it allows 'tests' to be performed on the models. This involves users observing general properties of the system through the graphical presentation of the models or performing specific tests such as equivalence- or incompleteness testing. The results of such tests can be shown graphically.

The models that are developed with *Mosel-MetaFrame* are based upon directed-graphs, an example of which can be seen in *Figure 2.1*.

Conclusion and Critique

The designers of *Mosel-MetaFrame* see visualisation as providing a way of rendering the formal specifications that are used to model their target systems as customer-centred representations. In addition, the designers also claim that their system is capable of improving the developer's experience by reducing the effort required to verify the correctness of the model.

These objectives are commendable, but the choice of representation used to depict the models can be challenged, especially in light of the fact that customers are part of the intended audience. It can be argued that the directed-graphs do not offer an adequate customer-oriented representation, and that much learning would still need to be done before customers could understand the resulting technical models.

Criteria	Description	Extent
Application Domain	Simulation of hardware circuits	
Visual cues	Geometric shapes from the domain directed-graphs	✓
Visualisation objects	Directed graphs symbols based on geometric shapes	✓
Scenes	Complete diagrams based on directed-graphs	✓✓
Dynamism	None	
Flexibility	Required to change underlying specifications to modify visualisations. Manual process.	✓
Representing Relationships	Possible to state position of entities within the diagram and connect them to others via arcs.	✓
Abstraction	Decomposition/expansion of diagram elements	✓✓
Correspondence	Correspondence between script and resulting visualisations exists, but is not entirely complete or comprehensive.	✓

Table 3.2. *Classification of the Mosel-MetaFrame visualisation approach.*

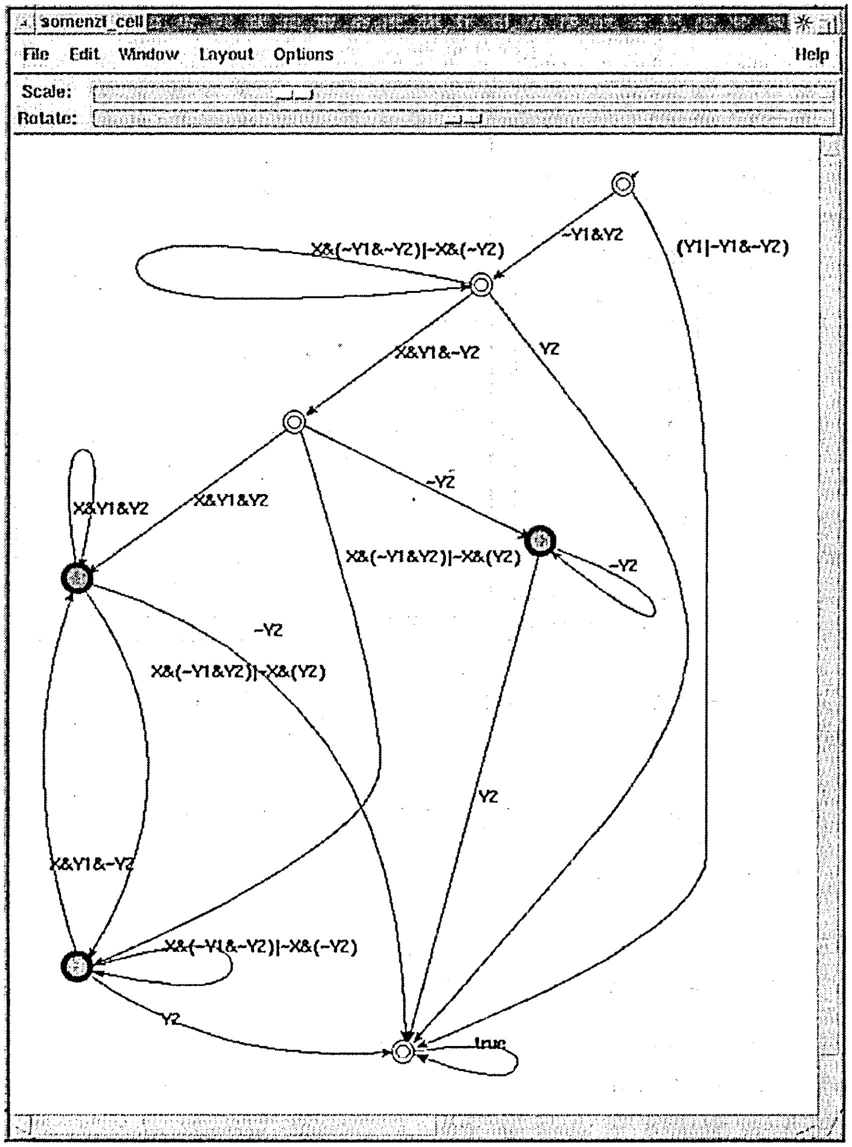


Figure 2.1. Example visualisation from the Mosel MetaFrame environment.

2.3.3 ENVISAGER [Diaz-Gonzales87]

ENVISAGER (ENvironment for the VIsual Specification And Graphical Execution of Requirements) has been designed to provide tools to support the conceptual modelling and simulation of real-time systems with the aim of using these in the requirements validation stage.

The objective of *ENVISAGER* is to support the modelling of the various characteristics inherent in real-time systems, including several independent (concurrent) processing entities or data sources/sinks which could be either software or hardware components, messages between these entities, synchronisation between different processes, sporadic events which must be dealt with, and response times that must be adhered to. To facilitate an accurate representation and simulation of a real-time system, knowledge about these characteristics must be included within the model.

The conceptual models developed in *ENVISAGER* are based upon a collection of objects that communicate through an asynchronous message passing mechanism. Logic predicates are used to specify and maintain knowledge about the individual objects that comprise the model and the interactions between them. The predicates are specified in terms of a proprietary formal specification language. The formal nature of these eliminates the potential for ambiguity and misinterpretation. Each object within a model has its own set of predicates as well as a state, or 'memory', which holds the current condition of that object. During execution of the model, the state of an object is changed depending on external events or messages from other objects. As well as a textual definition, each object has its own graphical representation, which may change during the course of execution to denote changes in its state.

These concepts and the underlying model descriptions are hidden behind a graphical interface that simplifies the development and execution of models. *Figure 2.2* shows the architecture of the system, which consists of a graphical specification editor, a static analyser, which is used to check the specifications for inconsistencies before execution, and a visual simulation tool to support execution.

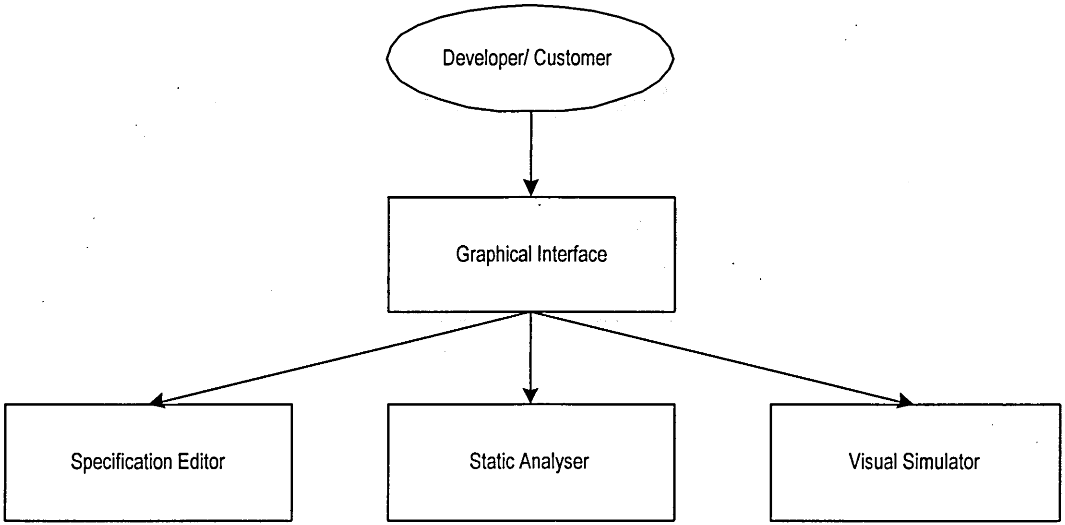


Figure 2.2*. The architectural composition of the Envisager system.*

The Graphical Specification Editor is a tool to facilitate the design and modification of model descriptions in a direct-manipulation style. Models are composed from graphical objects and icons which are all derived from graphics primitives such as lines, circles, and various fill patterns, etc. The editor also provides facilities for the specification of timing constraints.

A noteworthy aspect of the Specification Editor is the provision for reusability. Libraries of icons and graphical components for different application domains within the realms of real-time systems are maintained. The reusable components come with their associated text-based definitions that define their behaviours in terms of predicate logic.

The Visual Simulator provides the capabilities for executing a model, and presenting the changes made to the model during its execution to the viewer. The simulator also modifies graphical objects in accordance with changes to their internal state. The user can interface with the simulator by sending 'messages' to objects to initiate events in the system, or the system's environment. The system then reacts accordingly, visually displaying any changes to the model.

Conclusion and Critique

The *ENVISAGER* system provides a competent and effective system for the description and execution of models of real-time systems for the purpose of requirements validation. The system supports a variety of simple visual representations based upon graphical primitives.

However, for certain systems a simple visual representation may not suffice. The need for more advanced visual cue types, such as photographic pictures (in the form of bitmaps) may be required to produce visualisations that offer an adequate level of familiarity for customers. In addition, arguably the system does not provide adequate facilities for directly visualising the content of the internal states of objects, or the contents of messages, i.e. the data values that are encapsulated in them. Instead, the system provides a means of displaying a graphical view that merely changes in accordance with changes in state. Perhaps the major difficulty with *ENVISAGER* is finding a suitable graphical representation for every particular state or situation. This has been a traditionally difficult activity, as Rasure [Rasure91] points out "*It has been found, it is not a good idea to try to visualise everything. Some things, like mathematical formulas and numerical algorithms, are better handled with text.*" and as Tanimoto observes *"One of the most challenging aspects of designing visual languages is producing efficient graphical elements.*" [Tanimoto87].

Criteria	Description	Extent
Application Domain	Real-time systems.	
Visual cues	Icons, based on simple graphical primitives such as geometric shapes and lines.	✓
Visualisation objects	Icons representing specification elements that represent entities in real-time systems	✓
Scenes	Composition of icons representing complete depictions of real-time systems	✓✓
Dynamism	State changes in the executing model are reflected in the resulting visualisation as changes to the icons.	✓✓
Flexibility	Visualisations derived from text-based specifications, Must change the underlying to specification to cause changes to visualisations.	✓✓
Representing Relationships	Possible to state position of graphical objects and their adjacency to others. Viewers can infer relationships from this.	✓
Abstraction	Diagrams are abstractions/models of real-time systems, but no special mechanisms for abstraction available.	✓
Correspondence	The model is expressed/specified graphically, so a high degree of correspondence exists between the visualisations and the specification.	✓✓✓

Table 3.3. Classification of Envisager.

2.3.4 Visualising Concurrent Z Specifications [Evans94]

The approach presented in [Evans94] describes a system in which Coloured Petri Nets [Jensen81] are integrated with the formal specification language Z [Spivey92] to specify, prototype, and subsequently visualise a class of systems that are notoriously difficult to analyse and understand – concurrent systems.

The approach uses the Petri Net notation as a graphical specification method with which the developer can interactively construct models of (potentially) complex concurrent systems. The system provides facilities for automatically interpreting and converting these graphical specifications into Z specifications. From this, the reasoning power of Z may be used to prove safety, invariance, and timing oriented properties. *Figure 2.3* shows the process involved and the possible uses for the system.

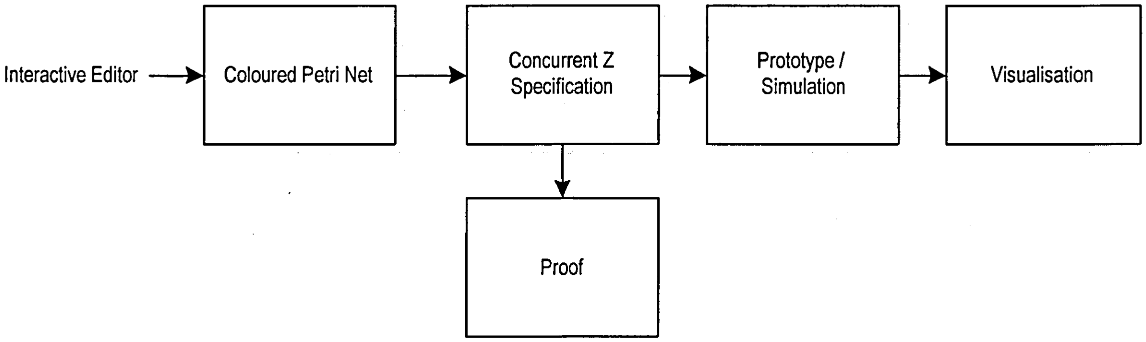


Figure 2.3. *The processes involved in visualising concurrent Z specifications.*

The specification technique employed in this approach, the Coloured Petri Net, is an enhancement to the traditional Petri Net notation, in that complex data structures can be attached to the tokens used in the models. This allows more complex models of concurrent systems to be specified using the notation. Predicates are also accommodated, so complex conditions can be modelled.

This approach hinges on the state-based nature of both Petri Nets and the Z notation. When a prototype is executed, changes in the state of the modelled system can be reflected in a visual representation of the model, thus giving the viewer a dynamic picture of the workings of the system under investigation.

The authors claim that the approach can be applied to other methods that include similar diagrammatic specification notations, such as State Transition Diagrams, Data Flow Diagrams, etc.

Conclusion and Critique

This system is targeted at concurrent systems, and therefore cannot be applied to a wide range of domains. Of concern however, is the inability of the system to provide appropriate visual representations that a customer can understand; only a single representation is offered, i.e. a Coloured Petri Net that may not offer a sufficiently customer-oriented view. The Coloured Petri Net notation has been designed to offer a way of modelling the complex structures, behaviours, and interactions inherent in concurrent systems, and is a tool used primarily by developers. It was not intended to reduce this complexity to a level where a customer could readily understand the behaviour of a system, which limits the potential of this approach for customer requirements validation, thus conflicting with the authors' original objectives. A similar system that visualises the execution of Petri-net based models can be seen in [Ae87].

Criteria	Description	Extent
Application Domain	Concurrent systems.	
Visual cues	Geometric shapes to form the symbols of Petri-Nets	✓
Visualisation objects	Graphical Petri Net elements, such as nodes and tokens	✓
Scenes	Complete Petri Net models, including tokens	✓✓
Dynamism	Shows real-time 'execution' of Petri-Net models.	✓
Flexibility	Changes in the Petri-Net model directly affect changes in the visualisation.	✓✓
Representing Relationships	Petri Nets are able to represent relationships between states of a system, but not special mechanisms available.	✓

Abstraction	Petri Nets are abstractions of real-time/concurrent systems. Individual Events can be broken down into lower-level nets.	✓✓
Correspondence	Petri nets are graphical by default, and the models have a high degree of correspondence between the graphical representations.	✓✓✓

Table 3.4. Classification of the Concurrent Z Visualisation approach.

2.3.5 Visualising VDM Execution [Cooling94]

The approach described in [Cooling94] outlines a system that is designed to visualise and animate formal specifications. The system is designed around the formal specification language VDM, and the target application domain is that of embedded real-time software systems. The objective of the approach is to "*illustrate the key properties of specifications to non-computer specialists.*"

The process of visualising VDM specifications using this approach is performed in two stages. The first stage concerns the development of a traditional VDM specification to model the system under investigation. The second stage involves the development of a text-based 'script' that drives the visualisation and animation sequences. The script is separate from the specification and contains the information necessary for visualisation. It can be thought of as a program that mimics the semantics of the underlying VDM specification and is executed to produce a visual simulation of its execution. The description of the approach, given in [Cooling94], describes the development of the visualisation scripts as a manual process.

The scripts describes both the static and the dynamic aspects of the visual representations, for instance the type, colours, and location of graphical objects and the elements that inform the support environment what to do in response to changes of the system state. Conventional programming constructs are provided such as conditions, iterations, and abstraction (via procedures). The system provides a variety of graphical primitives from which complex diagrams can be developed.

It is claimed that the system has been used to demonstrate formal specifications of systems to audiences including managers, engineers, and research workers, and their reactions have been most positive.

Conclusion and Critique

The system as described provides an adequate approach for visualising specifications based upon the formal specification language VDM. However, there are concerns as to the method used for constructing and executing visualisations. The development of visualisation scripts is a manual process. In this context, it is possible to construct scripts that do not correspond to the underlying VDM specification. Using this method it is possible for the advantages of formal methods to be lost, since ambiguity, misinterpretation and incorrectness can be introduced at the script level if correspondence does not occur.

If the script generation process is augmented with a method to produce visualisation scripts automatically, what should be the extent of the automation? Developing suitably expressive visualisations cannot be performed effectively by procedural means since visualisation is a creative activity, unless the visualisations are limited or constrained. As software has not yet gained the human qualities that are necessary for effective visualisation, some human intervention must take place.

An important additional point concerns the overall method of developing and subsequently modifying visualisations. Whichever approach is used, either manual or automatic, the resulting script is text-based. This approach has definite limitations with respect to changing the look and meaning of the visualisation. For example, suppose that part of the visual representation does not adequately reflect the VDM specification. It might be a time consuming and arduous task for the developer to have to re-write the appropriate portion of the script.

The use of two languages is also a concern. By forcing the developer to learn and use separate languages for the specification and visualisation components, difficulty and mental load is increased, introducing the potential for error.

Criteria	Description	Extent
Application Domain	Real-time software systems	
Visual cues	Icons representing elements in real-time systems	✓
Visualisation objects	Basic graphical primitives such as geometric shapes, and colours. Possible to construct diagrams from these.	✓
Scenes	Complete diagrams composed of icons	✓✓
Dynamism	Static visualisations only	
Flexibility	Visualisations derived from processing a text-based script. Changes must be made to the script, and then re-processed to modify visualisations.	✓

Representing Relationships	Possible to state position of graphical objects and their adjacency to others. Viewers can infer relationships from this.	✓
Abstraction	Resulting visualisations represent abstractions/models of real-time systems. Abstraction is also provided through procedures/functions for developing models.	✓✓
Correspondence	High level of correspondence between script and resulting visualisations. Problem with production of the underlying scripts – manual process, so it impossible to determine if these meaningfully depict the requirements.	✓

***Table 3.5**. Classification of the VDM visualisation approach.*

2.3.6 Problems with Existing Prototype Behaviour Visualisation Systems

From the review of existing requirements visualisation approaches above, five important shortcomings can be identified. These shortcomings form the rationale for developing an alternative visualisation approach, as it can be argued that they compromise the effectiveness of the systems – this is in spite of the good intentions of their creators.

The shortcomings can be characterised as lack of diverse or visually rich representations; lack of correspondence; difficulties with the development of visual representations; difficulties in associating visualisations to prototype definitions and execution behaviours; and the lack of a guide or method.

The lack of visually rich customer-focused representations in the systems presented above demonstrates how pertinent and real the problem of communication is, as discussed in *Section 2.2*, even though the systems attempt to overcome it using visualisation. It can be argued that these systems provide more examples of the application of technically oriented depictions, either textual or graphical, of prototype execution behaviour being used when customers are being involved in the prototyping process. In each case, it is not hard to imagine the difficulties faced by a customer when attempting to understand such representations.

A potentially serious problem that affects most of the above systems to a greater or lesser degree is the lack of correspondence between the meaning of the visual representation (that can be perceived by the viewer) and the meaning of the underlying prototype execution behaviour. It is vital that customers should understand the meaning that the developer intends if they are to make an informed judgement about the requirements. In some of the systems presented above, it may be argued that the connection between the visual representation and the prototype execution was unclear, i.e. a non-technical viewer might not establish how the visualisations portray the meaning of the behaviour of the prototype. This was due, in part, to the style and form of the visual representations available to depict execution.

Further, developing and specifying visual representations is a critical activity in any application of visualisation. Visualisation development involves two distinct steps. Firstly, the visualisation designer (which may be a software developer or a specially trained graphic artist for example) uses cognitive skills to imagine a conceptual representation. Secondly, an 'externalised' representation is developed that reflects the mental representation of the designer. This is achieved using a convenient or adequate notation or mechanism that enables the ideas to be expressed. It is desirable for this notation or mechanism to allow a representation to be expressed in a form that corresponds as closely as possible to the conceptual representation. This enables the designer's mental load to be reduced – it reduces the potential for error, improves productivity and reduces the constraints placed upon the creative process. Since the conceptual representation is arguably based upon a pictorial form, it is desirable for the notation to also be based upon a pictorial form. In other words, the objective is to describe visual representations using visual representations.

There are two main deficiencies in the approaches described above regarding the provision of support for developing visualisations. The first is the use of an inappropriate means of defining visual representations. Some of the approaches employ a textual language or scripting technique. Whilst accurate and concise, such approaches constrain the designer of the visualisations in two ways. Firstly, such languages do not correspond to the visual designer's conceptual representations, and secondly, they force the designer into learning an additional notation to perform the externalisation process. The second deficiency is the lack of tool support. This is critical in the development of visual representations. Tools assist the visualiser in externalising the conceptual representation. However, in some of the approaches described above, tool support is minimal, providing only limited editing facilities. In some cases, tool support consists of a text editor, by which visualisation scripts or the predicates that attach visual representations to specification components can be edited. An important issue also lies with how to associate the resulting representations with the requirements models. This particular problem stems from the characteristics of a prototype, in that it is a description of the functionality of a software system. Ideally, this description should only include the details of the functional behaviour and exclude any extraneous information that could confuse the meaning of the requirements. This is especially pertinent when using formal specification techniques in a dual role – for generating prototypes (through specification execution) and as the SRS. A potential source of pollution could be the inclusion of the information required to perform visualisation within the requirements model itself. This situation could lead to confusion as to what aspects comprise the model and what aspects comprise the visualisation details. To prevent this, the visualisation system should limit the amount of detail required for visualising the specification and facilitate the expression of such detail in a minimal form, hiding any internal structure or content. Some of the approaches described above do not make a clear separation between the prototype definition and the definition of visual representations. Instead, they force a developer to associate large amounts of visualisation details in specifications.

The last major deficiency that can be identified with existing systems is the lack of accompanying method or process. Software engineering is replete with methods and processes. These are defined prescriptive guides for developers and serve to describe and communicate the tasks for achieving a given goal. The aim of any method is to provide suitable guidance to render potentially complex procedures as a simple and easy-to-follow series of steps. However, many of these approaches lack such methods. The mere provision of software tools is not sufficient. Guidance is often necessary which will inform the developer of the optimum and effective strategy for using the toolset. In addition, if followed correctly, such a method would ensure that no activities are omitted, leading to a more successful process.

2.4 Summary

This chapter has elaborated upon the context for the research that is presented in the remainder of this thesis. It described a problem that can impede the effectives of the popular and important requirements validation approach, namely prototyping. The problem concerned customers being unable to comprehend prototype behaviour due to its execution being presented using representations that are not customer-centred. A potential solution was then advocated based upon the use of visualisation. The solution advocated that familiar customer-oriented visual representations could be substituted as alternative representations, so increasing the comprehensibility of prototype behaviour.

Visualisation was promoted as having the apparatus necessary to deliver such representations. A survey of related work was then undertaken. This looked at various efforts to visualise prototype execution behaviour. The aim was to identify deficiencies to provide a motivation for the research.

The next chapter builds upon this. It describes the requirements and design for an alternative prototype execution visualisation system that aims to overcome the deficiencies found with existing systems.

Chapter 3

An Alternative Prototype Execution Visualisation System

The previous chapter discussed the advantages of visualising the prototype execution in the context of requirements engineering. It reviewed a number of existing approaches to prototype execution visualisation and identified some important shortcomings with these, which included the lack of rich and diverse visual representations, the lack of correspondence between the meaning of visual representations and the prototype, and the lack of a process or guide for users of the approach.

This chapter describes in detail a prototype execution visualisation system named **ViZ** which provides an enhanced environment for visualising the execution of a specific prototype execution technique. Its aim is to facilitate customer/user validation. In doing so, this chapter presents the main contribution of the work.

ViZ comprises a lightweight process and a software based toolset. The process describes the steps necessary for developing visualisations of prototype execution; it also acts as a guide for the toolset of the ViZ system that provides the visualisation capability.

The ViZ system is described in three sections. The first presents an overview of the system, and shows its major characteristics and properties. The second describes the details of the ViZ process, presenting the stages that comprise the process, the work products and the deliverables that result. Lastly, the third section presents details of the software toolset, including its capabilities and component architecture.

3.1 System Overview

This section presents an overview of the ViZ system. It is divided into three parts. The first describes the context in which the system is developed by describing the characteristics of a prototyping approach to which visualisation will be applied. The second part describes a set of requirements that the ViZ system should fulfil to visualise the execution of prototypes developed using this chosen prototyping approach. The final part of the overview describes the specific characteristics of the VIZ system.

3.1.1 System Context

Realising the research objectives – demonstrating how visualisation may be applied to prototyping – may be achieved in one of two ways. The first approach involves the development of a new prototyping system simply as a vehicle for the research. The visualisation facilities would then be developed in tandem with this, and would be integrated closely with the prototyping approach. The second approach is to select an already existing prototyping system and augment visualisation facilities to this.

For the purpose of this research, it is this second approach that is taken, whereby an existing prototype system is used as the vehicle for demonstrating the research goals. This approach was taken to alleviate the effort required to develop a new prototyping technique. However, this strategy is not without its difficulties. The characteristics of the existing system must be explored fully in order for visualisation to be integrated successfully. To this end, this section examines the chosen prototyping approach.

Known as *RealiZe* ("Requirements engineering by animating Z extensions"), this prototyping development environment aims to improve the quality of requirements specifications through the application of formal methods [Morrey98, Buckberry99, Hibberd01]. This approach, developed as the product of research by the Requirements Engineering Research Group (RERG) within the School of Computing and Management Sciences at Sheffield Hallam University, provides an execution environment and usage process for the popular state-based requirements specification language Z [Spivey92]. A software developer can use RealiZe to specify the functionality of a software system and then animate it, resulting in the production of a prototype that depicts the specified system's functional behaviour. This prototype can then be evaluated or subsequently modified in the context of requirements validation.

The RealiZe system comprises two complementary software tools. The first is *TranZit*, which offers full-screen WYSIWYG-style editing, syntax analysis, and type checking facilities for Z specifications, as well as a transformation engine that allows Z specifications to be converted into an executable form. This form is a notation based upon Common Lisp, which is processed by the second tool in the RealiZe system – *ZAL*. ZAL (Z Animation in Lisp) is the name for both the notation and the tool. The tool embodies mechanisms necessary to execute the specifications that are expressed with the notation. The relationship between the tools is that TranZit is used by developers to construct and format Z specifications, and subsequently transform these into ZAL specifications, then produce a prototype by executing the specifications with the ZAL run-time support system. The two tools and their relationship to developers and customers are shown in *Figure 3.1*.

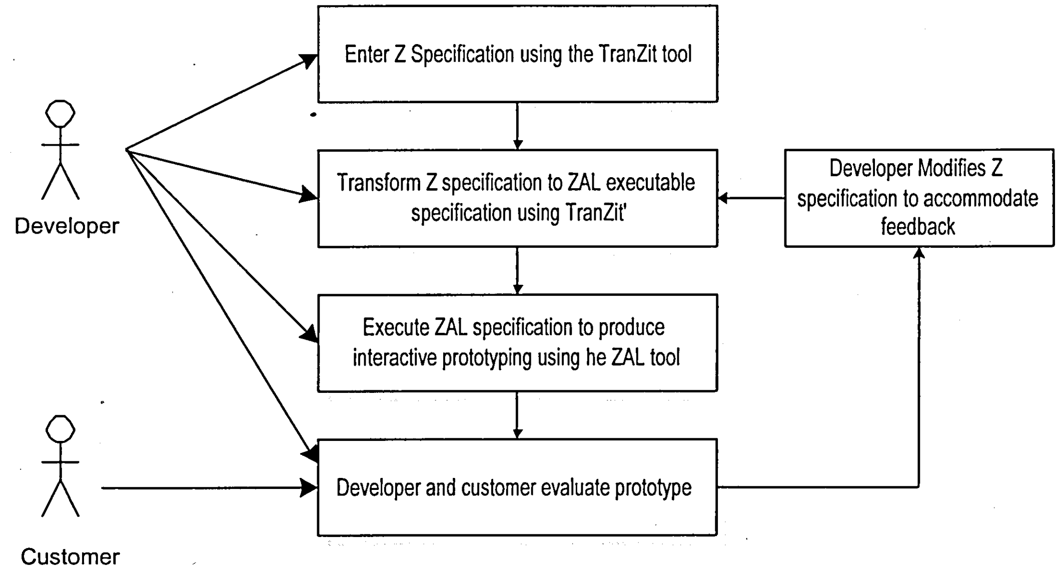

Enter Z Specification using the TranZit tool
Transform Z specification to ZAL executable specification using TranZit'
Developer Modifies Z specification to accommodate feedback
Execute ZAL specification to produce interactive prototyping using he ZAL tool
Developer and customer evaluate prototype
Developer
Customer

***Figure 3.1**. The relationship between the tools in the RealiZe Approach*

One of the key features of the RealiZe approach is its ability to transform a captured Z specification into a representation that facilitates execution, through the TranZit tool [Buckberry99]. This is a non-trivial process, as one of the characteristics of Z is that many of its constructs do not lend themselves to direct execution [Hayes89]. Therefore, the transformation process is required to translate Z-based specifications into forms that are directly executable yet still offer a large degree of semantic equivalence. While comprehensive, in that a considerably large subset of the Z notation is accommodated in ZAL, a small number of modelling constructs are not offered due to the inherent difficulties in finding equivalent executable forms. For these constructs, the developer must intercede and provide alternatives.

In terms of ZAL prototype execution, a characterisation can be found from an analysis of the two important aspects of such prototypes, namely notation and behaviour. In terms of notation, prototypes are defined as a series of ZAL-based expressions that describe functional behaviour. In addition, a data or state description can also be specified. This comprises a set of data stores, in the form of system state variables, with each one consisting of an identifier and value.

A relationship between the descriptions of behaviour and state exists. Expressions that represent behaviour consist of an operation and a set of associated arguments. The arguments, in the form of identifiers, refer to variables that are a subset of the system state. These are processed or transformed as an operation is evaluated by the ZAL support system. Each expression, along with its arguments can be described as:

```
<expression>     ::=    <operation> <argument set>
<argument set>   ::=    [<argument>]
<argument>       ::=    <variable identifier>
```

In addition to the provision of behaviour and state, ZAL enables data inputs to be defined and used in the execution of specifications. These provide an opportunity, at run-time, for a user to enter data into variables that will be processed by expressions. Moreover, ZAL facilitates the display of outputs, which can be used to show the contents of system state variables or messages at run-time.

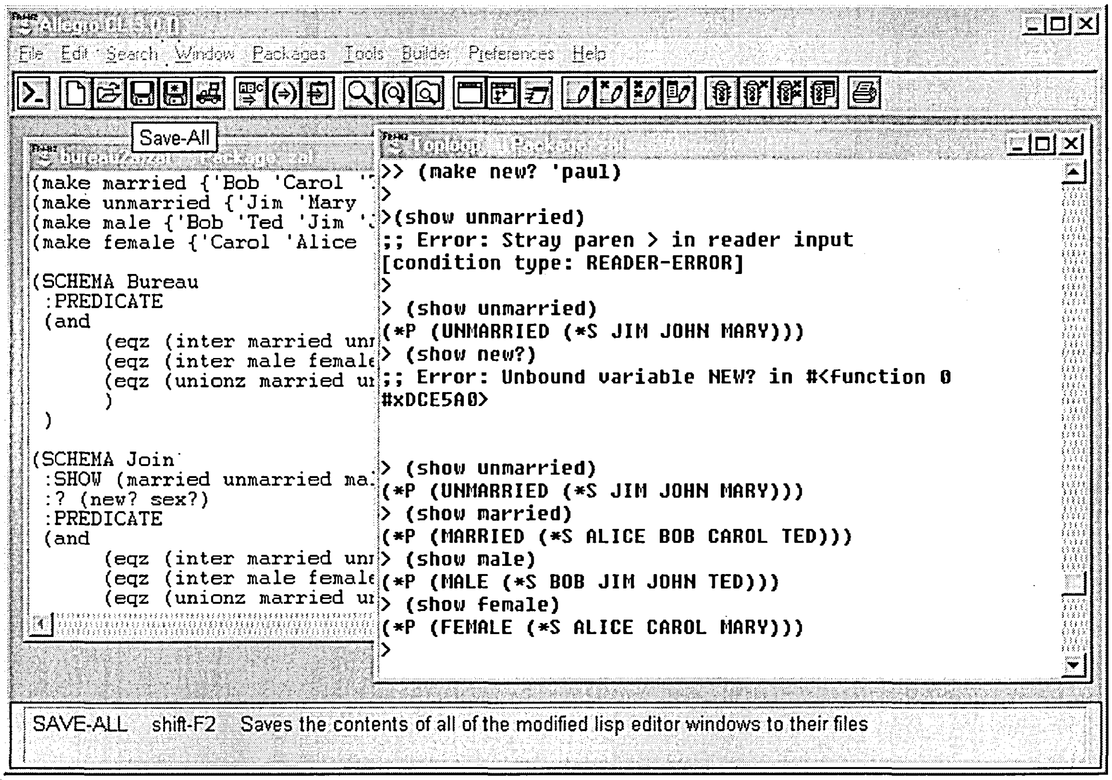
The evaluation of the expression produces a result, this being a modified system state, or an output/message. Along with inputs, these items may be termed *execution artefacts* since they are derived from or are directly involved in execution.

Execution can be viewed as a state machine, whereby it is seen as a series of discrete states and changes to these states. The initial state represents the state of the prototype (i.e. the configuration and content of the system state variables) before an expression is evaluated. A discrete event then takes place to denote the expression being evaluated and the system state being modified, if applicable. A second state then exists that represents the new state of the prototype. This series of state changes continues until the final expression in the prototype definition is evaluated.

To commence execution, a user, via a text-based command-line interface, invokes a particular schema and supplies any input data. Execution of that schema is conducted by processing each predicate in the schema sequentially. Execution of each predicate can be thought of as a transformation process, whereby operands are modified in accordance to the semantics defined by the operation. The result of this is the modification of the system state or the generation and display of outputs.

ZAL does facilitate a simple display mechanism for execution artefacts, but this is based upon a rudimentary text-based approach that presents items in the form of the underlying LISP data constructs. *Figure 3.2* shows a typical snapshot of the ZAL environment and illustrates the user interface. This particular example shows the execution of a simple database that is used to store details of the marital status of individuals.

Despite the apparent utility of the ZAL system, and the ease of prototype development through formal specification, the ZAL execution engine does not offer expressive or richly interactive methods of rendering the prototype's behaviour. It can be argued that this style of interface does not facilitate effective user validation. This user interface offers a substantial obstacle with regards to how a developer or customer might interact with the prototype, invoke its functionality, or provide input data that is often required to facilitate execution. It prevents stakeholders engaging fully with the prototype, and provides practical barriers to comprehending its operation, its progress, or the contents of system state variables.



```
(make married {'Bob 'Carol ':
(make unmarried {'Jim 'Mary
(make male {'Bob 'Ted 'Jim ':
(make female {'Carol 'Alice

(SCHEMA Bureau
 :PREDICATE
 (and
      (eqz (inter married unr
      (eqz (inter male female
      (eqz (unionz married ur
      )
 )

(SCHEMA Join
 :SHOW (married unmarried ma:
 :? (new? sex?)
 :PREDICATE
 (and
      (eqz (inter married unr
      (eqz (inter male female
      (eqz (unionz married ur
```

```
>> (make new? 'paul)
>
>(show unmarried)
;; Error: Stray paren > in reader input
[condition type: READER-ERROR]
>
> (show unmarried)
(*P (UNMARRIED (*S JIM JOHN MARY)))
> (show new?)
;; Error: Unbound variable NEW? in #<function 0
#xDCE5A0>


> (show unmarried)
(*P (UNMARRIED (*S JIM JOHN MARY)))
> (show married)
(*P (MARRIED (*S ALICE BOB CAROL TED)))
> (show male)
(*P (MALE (*S BOB JIM JOHN TED)))
> (show female)
(*P (FEMALE (*S ALICE CAROL MARY)))
>
```

SAVE-ALL shift-F2 Saves the contents of all of the modified lisp editor windows to their files

***Figure 3.2**. An illustration of the ZAL environment.*

In addition, the effects of the user interface are compounded by the complexities introduced by formal methods. To the customer, formality presents another barrier to comprehension. The ZAL system's formal roots are reflected in its user interface where the progress of execution, depicted by displaying the contents of system state variables and outputs, are displayed using predicate or set-based representations, or other terse messages.

Despite possessing several major benefits for the developer (such as a swift prototype construction and modification tool, and in parallel, the production of unambiguous specifications), it can be argued, on the basis of facilitating prototype comprehension, that it is limited in the support it offers to the customer while undertaking requirements validation. For this reason, the ZAL environment offers a suitable candidate to which visualisation can be applied. Moreover, by complementing this particular prototyping system with visualisation, a requirement validation suite can be developed that provides support for the complete validation lifecycle.

3.1.2 Requirements of the ViZ System

The aim of the ViZ system is to provide comprehensive visualisation capabilities and support for visualising the execution of ZAL-based prototypes. As such, it is necessary to define a set of requirements that the system should fulfil. These requirements, elaborated below, concern three areas of provision, namely visualisation provision, software integration and process support.

In terms of traceability, these requirements are derived from a variety of sources. The first source is the critical factors that pertain to presenting prototype execution behaviour (described in *Section 2.2.1*). The second is the general issues in applying visualisation to prototype execution *(Section 2.2.2*). Combining these lead to some specific challenges being defined (*Section 2.2.3*). The third source includes the deficiencies identified with existing systems (elaborated in *Section 2.3*). The fourth and final source is the need to accommodate the characteristics of the ZAL system (described in *Section 3.1.1*). These sources are summarised in *Figure 3.3*.

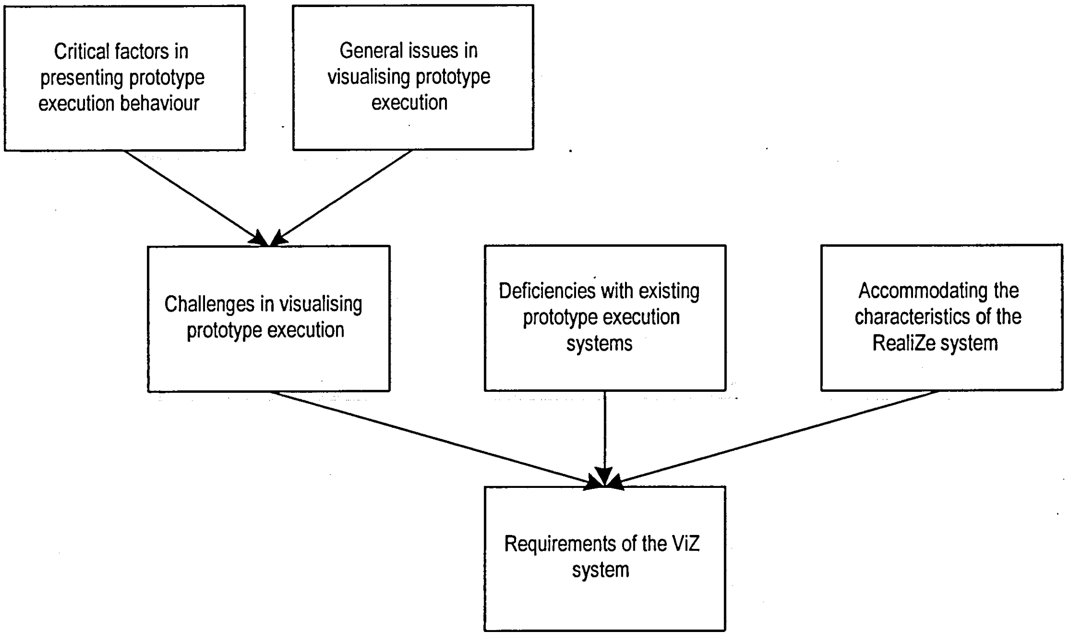

Critical factors in presenting prototype execution behaviour
General issues in visualising prototype execution
Challenges in visualising prototype execution
Deficiencies with existing prototype execution systems
Accommodating the characteristics of the RealiZe system
Requirements of the ViZ system

***Figure 3.3**. Source and traceability details of the requirements for the ViZ System.*

Visualisation Provision

Visualisations are central to the effectives of the approach. It is necessary therefore, to provide the range of visual apparatus in accordance with the visualisation issues described in *Section 2.2.3*. To this end, the approach should provide a range of individual visual cues, and then enable these to be combined into visualisation objects that can be applied to represent execution artefacts or contextual details. Subsequently, the approach should facilitate combining these into scenes that visualise major events, such as the execution of an expression, at run time.

These fundamental qualities should be augmented with capabilities concerning the creation and subsequent management of visual representations. It is necessary for facilities to be available for developing, storing and modifying visual representations.

As a consequence of the requirements validation process, a prototype might be built rapidly and then undergo significant change. Reuse of visual representations is therefore an important requirement to facilitate a timely visual prototype development and evaluation cycle.

Features that assist in preserving integrity between the meaning of visualisations and meaning of specifications are also of vital importance. This addresses the correspondence problem that was identified with some of the existing visualisation systems. It is important that the visualisation system possesses mechanisms to constrain the possibility of introducing ambiguity into the validation process. For reasons expressed in *Section 2.2.3*, a complete solution to this may be unreachable. However, a compromise may be found between freedom and constraint by a 'certification' approach, whereby visual representations could be evaluated and classified as being suitable for a particular purpose by software developers or domain experts before the prototyping process commences.

Software Integration

These requirements concern the practical and technical aspects of integrating visualisation software with the ZAL tools.

First, there is a need for a separate tool, working in concert with ZAL. The arrangement should be such that ZAL would provide the underlying execution support and ViZ would provide the necessary visualisation capability. A separate tool is required as the ZAL system is an already existing self-contained software entity. Consequently, there is a requirement for the two tools to interoperate so visualisation can be performed. This, in turn, requires the tools to communicate along with suitable protocols to facilitate this.

Second, it is desirable to employ a mechanism that is efficient and minimal with regards to associating visualisation details with a prototype definition, so they can be interpreted and rendered at run-time. If a strict separation of concerns is maintained, then identifying the respective visualisation details and prototype definition is straightforward – this is advantageous when developing or modifying either.

Process Support

A process is required to guide the users of the approach through the activities of developing visual prototypes. Such a process would facilitate repeatability, in that it could be applied consistently to subsequent development projects, and not leave prototype construction and use in the domain of ad-hoc development. As such, it should define, from the perspective of the stakeholders, the activities of identifying which aspects of the prototypes should become the focus of visualisation, developing suitable visualisations and associating them with these aspects, and executing and evaluating the resulting prototype.

The process should also combine the activities that comprise the RealiZe process with those responsible for visualisation development.

A complete set of requirements documentation, encapsulating system and software requirements are given in *Appendix A* and *Appendix B* respectively. Both specifications are structured and presented in accordance with the relevant IEEE standards on documenting system and software requirements [IEEE98a, IEEE98b]. The system specification takes the view that the ViZ system is integrated closely with the TranZit and ZAL approaches, providing a seamless environment for prototype definition, execution and visualisation. As such it documents the facilities that are available across all three systems. The software specification, however, focuses on the features that pertain only to the ViZ software system.

3.1.3 ViZ System Characteristics and Capabilities

ViZ (Visualisation of Z) provides an approach for validating ZAL specifications via specification execution and visualisation. The characteristics of ViZ are such that it fulfils the requirements of visualisation provision, ZAL integration and process support that were described above.

The objective is to retain the utility of the existing RealiZe approach and tools to provide the basis for prototype construction and use, but attempt, through visualisation, to overcome the difficulties in presenting execution behaviour. This is achieved through an execution-driven visualisation mechanism. To this end, ViZ integrates closely with RealiZe. *Figure 3.4* illustrates this integration and the responsibilities and scope of each of the processes and tools involved. The common factor between the three tools is the ZAL specification. The existing RealiZe approach is used to develop and furnish prototype behaviour through the development and execution of a ZAL specification, whereas ViZ provides the means to visualise its execution.

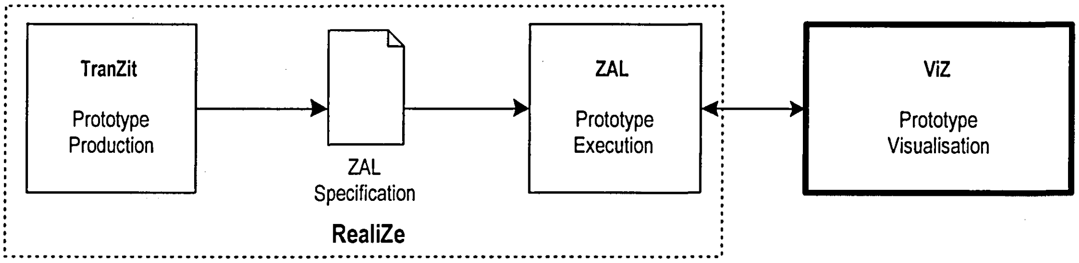


***Figure 3.4**. Relationships and responsibilities between the RealiZe and ViZ processes.*

ViZ consists of two related components: a process that forms the basis for visualisation development and application, and a software tool that supports the process by providing the necessary visualisation capability. These two components form the substantive and novel contribution of this research.

In terms of fulfilling requirements (as described previously in *Section 3.1.2*), the process fulfils the requirement of process support, whilst the software tool fulfils the requirements of visualisation provision and software integration.

The ViZ process documents a 'workflow' that describes the steps a developer or visualisation designer must undertake, using the ViZ software, to create and apply visualisations to a ZAL-based prototype. The process aims to impart structure over the potentially ad-hoc activities of visual prototype development and use. As such, it is argued that the process represents a significant step in promoting a repeatable visual prototype development method.

The ViZ software toolset [Parry00] complements the process. It can be described in terms of an architecture that is divided into three components. The first component is responsible for realising visualisation provision. The second is concerned with integrating visualisation with ZAL execution, whilst the third, the visualisation engine, acts as a bridge between these to provide co-ordination and synchronisation of specification execution and real-time rendering of visualisations. An overview of the tool architecture and the interrelationships to external software systems is shown in *Figure 3.5*.

Having provided a brief overview of the capabilities and features of the ViZ approach, the process and associated toolset are described in greater detail in the following sections.

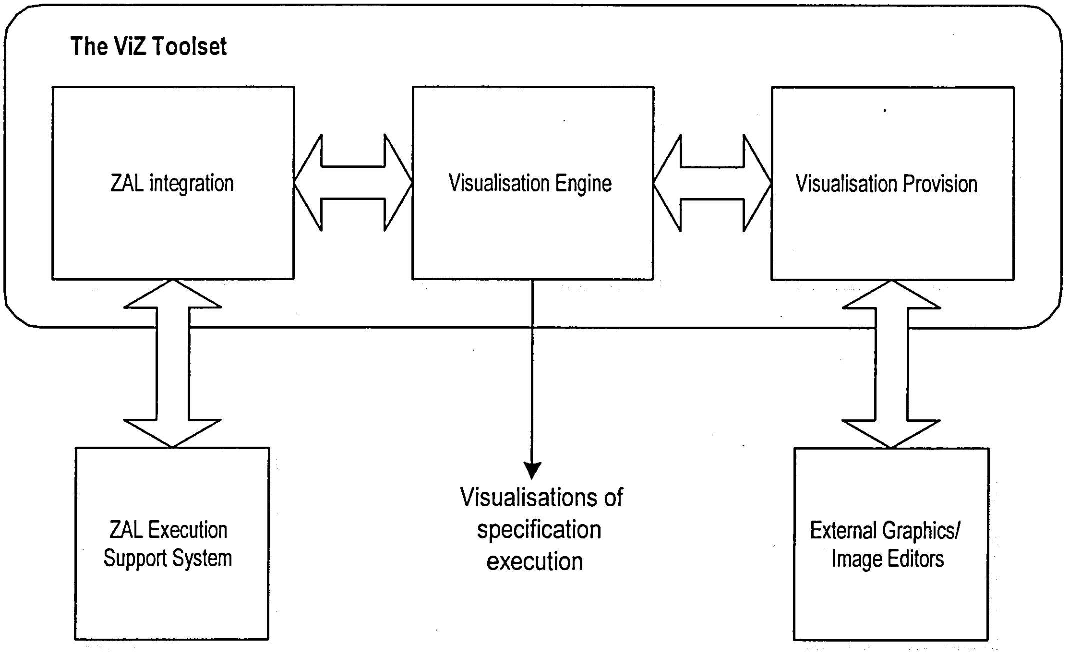

The ViZ Toolset
ZAL integration
Visualisation Engine
Visualisation Provision
ZAL Execution Support System
Visualisations of specification execution
External Graphics/ Image Editors

Figure 3.5. Overview of the ViZ toolset.

3.2 The ViZ Process

The ViZ process dictates a comprehensive step-by-step approach for visual prototype development and application. In addition, the process addresses three specific requirements:

- To provide general guidance for stakeholders involved in visual prototype development.
- To facilitate process integration between the RealiZe and ViZ approaches.
- To provide the basis of the correspondence preserving mechanism.

The process commences from the point where a ZAL specification has been developed using the RealiZe approach. The first step entails identification and documentation of descriptive scenarios. Scenarios sit well with the ViZ process and have multiple roles within it. They are used to provide a means of eliciting requirements, and they are used to indicate 'test cases' which can then be used to guide prototype evaluation.

Importantly, scenarios assist directly in the second stage of the ViZ process, which concerns the design and development of visual representations. By exploiting a relationship between the structure of scenarios and the structure of a Z specification, scenarios can be used to indicate elements in a specification that should become targets for visualisation. During this stage, and using the ViZ toolset, a visual prototype is constructed.

Although scenarios can be used to indicate targets for visualisation, they do not provide a means to assure that the meanings of the visual representations developed to portray them do indeed correspond to the meanings of the targets. To this end, the process provides the basis of a correspondence preserving mechanism that prescribes the 'certification' of visual representations by expert stakeholders as being suitable for portraying the targets being considered. This certification is performed during or prior to the development of a visual prototype, whereby appearances are developed and certified before they are applied.

The third and final stage in the process is prototype execution and evaluation. Here, the prototype is evaluated in accordance with test cases derived from the scenario descriptions. The aim is to involve the relevant stakeholders to stimulate discussion about the requirements that the visual prototype represents.

This series of actions is repeated until an agreement is reached as to the accuracy of the requirements. A characterisation of the process is shown in *Figure 3.6*. This shows the three stages along with the steps inherent in each stage. *Figure 3.6* also shows the relationships between the developer, the customer and the process, and where each interested party is involved. A detailed elaboration of each stage is given below.

3.2.1 Scenario Identification and Documentation

The first stage in the ViZ process concerns the identification and documentation of scenarios. This stage is required since scenarios play a key role in the ViZ process, and is performed by the developer, with potential assistance from the customer. The usefulness of scenarios in the ViZ process stems from three relationships that can be identified and exploited. The first is between the structure of a scenario and the structure of a Z/ZAL specification. The second is between scenario and the scenes that comprise a visualisation, and the third is between scenes and scenarios. These relationships are summarised in *Figure 3.7*, and are such that the expressions in a Z/ZAL specification model the activities in a scenario, and the activities in the scenario equate to scenes in the visualisation.

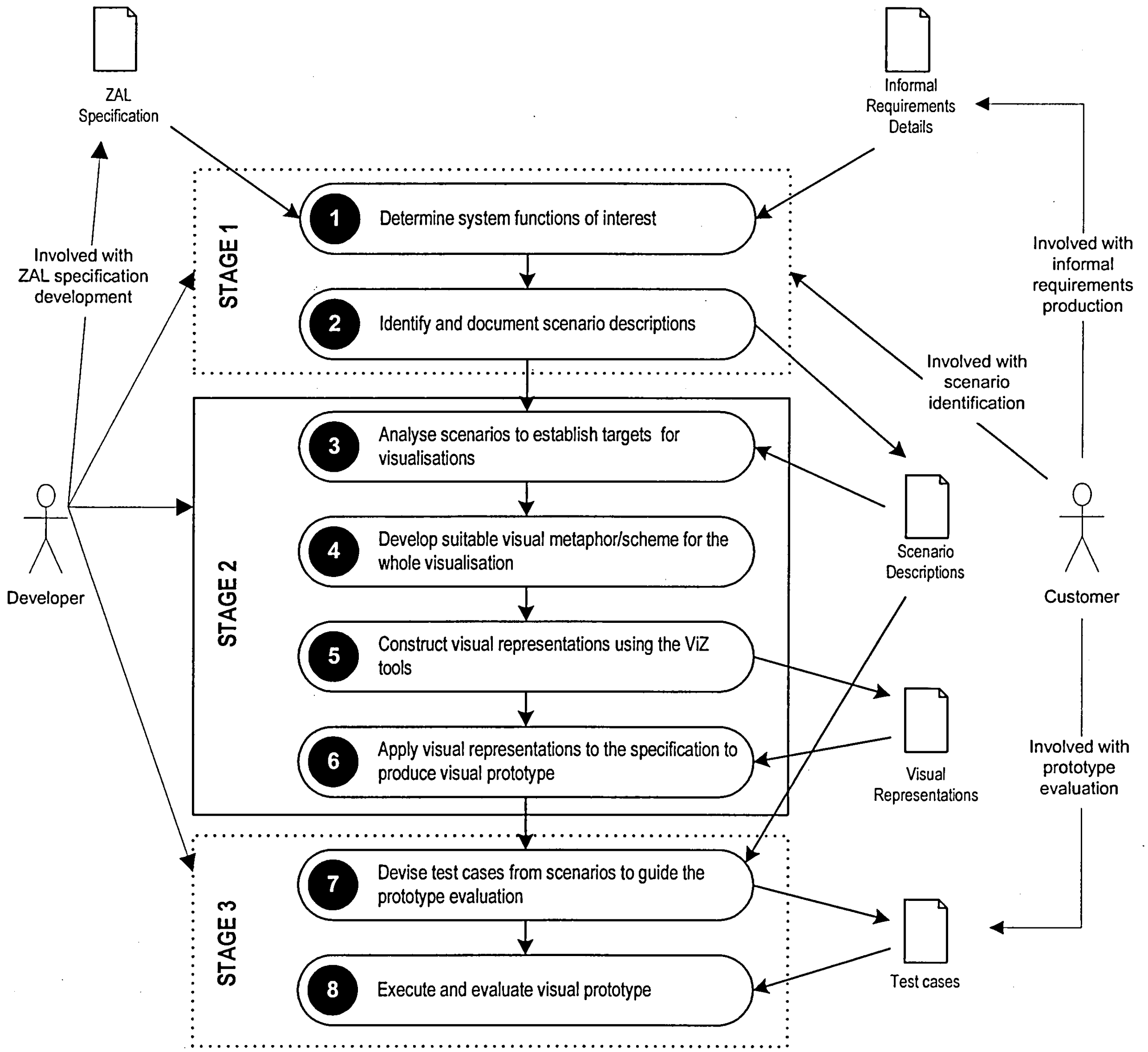

ZAL Specification
Informal Requirements Details
Involved with ZAL specification development
Involved with informal requirements production
Involved with scenario identification
Involved with prototype evaluation
STAGE 1
STAGE 2
STAGE 3
1 Determine system functions of interest
2 Identify and document scenario descriptions
3 Analyse scenarios to establish targets for visualisations
4 Develop suitable visual metaphor/scheme for the whole visualisation
5 Construct visual representations using the ViZ tools
6 Apply visual representations to the specification to produce visual prototype
7 Devise test cases from scenarios to guide the prototype evaluation
8 Execute and evaluate visual prototype
Developer
Customer
Scenario Descriptions
Visual Representations
Test cases

***Figure 3.6.** Overview of the ViZ process for visual prototype development and use.*

The relationships enable the developer or visualisation designer to identify the following aspects of a visualisation:

- The general structure of the visualisations, in terms of scenes that are required to portray the execution of the schema.
- The content of the scenes, i.e. the execution artefacts and contextual details that should form part of the visual prototype.
- The expressions in the specification to which scenes should be attached, so the visualisations can be rendered appropriately as the expressions are interpreted by the ViZ system at run-time.

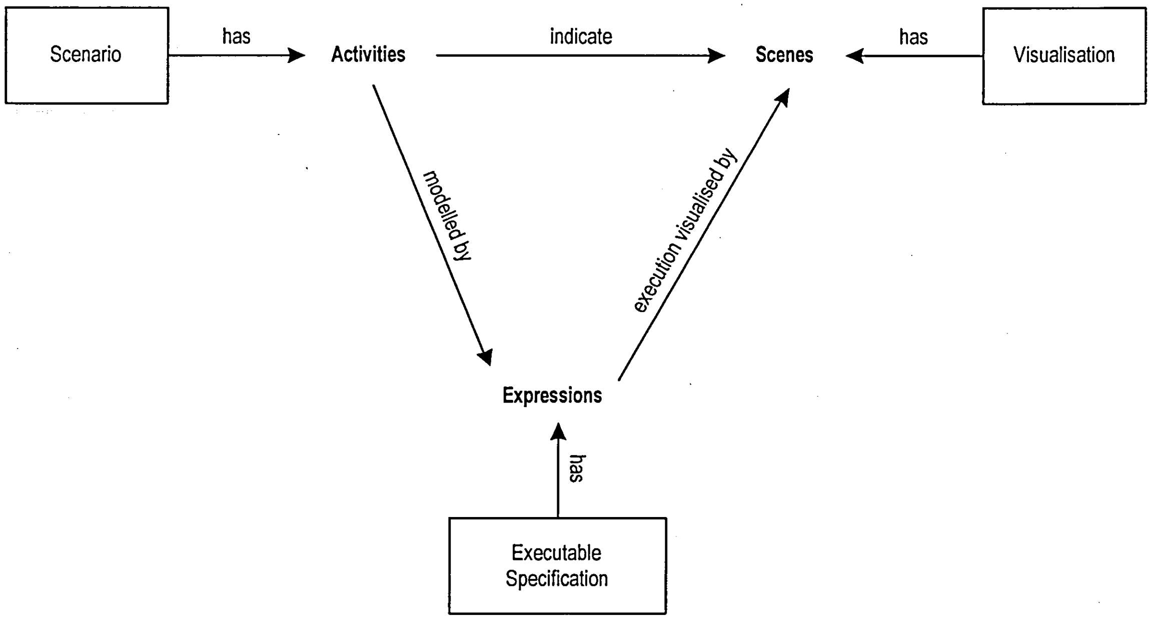


Figure 3.7. *Relationships between scenarios, visualisations and specifications.*

Hence, the approach provides a purposeful basis on which visualisations can be developed. It alleviates random or ad-hoc selection of specification elements to visualise – some of which may be necessary but omitted, while others may be present but contribute little to the resulting visualisation.

Scenario descriptions can be derived from the ZAL executable specification, the underlying requirements, or any other information gathered from the requirements engineering activities prior to specification production. It may even be the case that the requirements have been described using scenarios and a complete set of documented scenario descriptions already exist. In this situation, the scenario identification and documentation stage may be omitted.

Scenario identification involves two steps: 1) identification of the system functions of interest, and 2) scenario identification and documentation. These steps are examined below.

Scenario Identification - Step 1

The first step is to identify the system function (or functions) that are to be validated or that are to become the focus of further investigation. This step entails the identification of schemas in the ZAL executable specification that represent these functions.

Scenario Identification - Step 2

Scenarios that illustrate how the functions established in the first step are used are described in detail. This requires a list of scenarios that are of interest to be defined then their contents to be elaborated. Establishing the contents of the scenarios is performed by inspecting the expressions in the chosen schemas. The purpose of each expression in the given schema is elicited along with its relationship to the user. The requirements of the system (i.e. the unformalised description) can be instrumental in this. They enable the developer to determine the context of the expression and identify its purpose with respect to the overall system.

Scenario descriptions need to be expressed in such a way as to make the constituent parts of a scenario visible and easily identifiable. This is so the descriptions are indeed useful in the subsequent stages of the ViZ process. To facilitate this, the ViZ process borrows a useful technique from an existing scenario-based approach, originally proposed by Regnell [Regnell95]. This technique is a diagrammatic notation for representing scenario descriptions. It not only enables the actors and their actions to be expressed, but also enables data inputs and system responses/outputs to be defined. Moreover, in Regnell's notation, the sources and destinations of data (other than actors) can also be made explicit. Such entities are termed Abstract Interface Objects (AIOs). These form the interfaces between the system and its outside environment, and include screens, keyboards, etc. AIO's indicate a further aspect of the system that should be visualised that corresponds not to elements in a specification but to contextual details.

Several scenario descriptions might be required to express the variety of situations that may arise when a system function is invoked. These may describe normal uses cases or exceptional cases where the system under consideration must respond to erroneous inputs for example.

However, scenario identification and documentation is not without its difficulties. One of the most important aspects of a scenario – actors – are sometimes not readily identifiable. The problem stems from the difficulty in interpreting the definition of 'actor' – for some systems, the standard definition of an actor, i.e. a user or other external entity that invokes a system function or is a recipient of the system's output, does not necessarily fit exactly. Such difficulties may occur, for example, if users or external systems are not specified overtly in the requirements, or are otherwise assumed to exist. In addition, identifying abstract interface objects can also be problematic. The requirements for some systems may not directly specify interface entities. However, if either actors or interface objects cannot be identified easily, this may indicate a more fundamental problem with understanding the underlying requirements, requiring further discussion with the system's stakeholders. As such, in addition to forming a foundation upon which to base visualisation development and specification execution, this stage in the ViZ process can also be perceived as the first line of defence against erroneous specifications. Scenario documentation provides an early opportunity, without specification execution or visualisation, to identify and correct omissions or errors in a specification.

3.2.2 Visualisation Design and Construction

The second stage of the ViZ process aims to produce a visual prototype. This involves the design and construction of visualisations for this purpose. This stage in the process adds structure to the potentially ad-hoc nature of visualisation design. The key driver is the need to develop visual representations that are suggestive of the meaning of the underlying formal specification or requirements description.

This stage comprises four steps: 1) scenario analysis, 2) visual metaphor design, 3) design and construction of visual representations, and 4) application of visual representations to a specification to produce a visual prototype. These steps will be explored in detail below.

Visualisation Design and Construction - Step 1

An analysis of the scenario descriptions that result from Stage 1 comprises the first step in Stage 2. This analysis exploits the relationships (described above) between scenarios, Z/ZAL specifications and visualisations. The intention is to extract details from the scenarios that direct the development of visual representations. To this end, the relationships are used as follows.

Firstly, the relationship that is exhibited between scenarios and visualisations can be used to determine the overall structure, in terms of scenes, of the visual prototype. The structure may be thought of as a framework that provides basic information about the number of scenes required to portray the scenario activities and the sequence in which they should be presented. The relationship specifies that for each activity in a scenario one scene is required. This structure provides a starting point for further visualisation development. In practice, there may be a variety of subtly different views of the structure. However, the aim at this point is to provide an indication of the possibilities with regards to visualising the execution of the specification, and not to produce concrete visualisations this early.

Secondly, exploiting the relationship between scenarios and specifications can indicate the content of the scenes. This works as follows. Effective visualisation of prototype execution requires visualisation objects be developed to portray certain execution artefacts, namely data stores and inputs/outputs (as modelled by system state variables). It can be seen that at the activity level, scenarios possess certain elements that closely correspond to the execution artefacts, such as data values and inputs/outputs. By performing a simple analysis of a scenario, as expressed using the Regnell notation, such elements can easily be discerned. By using scenarios in this way, it is possible to establish accurately the visualisation targets. It is argued that this method provides a more straightforward approach, and one that is less likely to be prone to error, than simply searching through formal specification definitions.

Importantly, to provide contextual details for the visualisation of data, AIOs and actors can be identified as targets for visualisation. These elements can only be established from the scenarios, since formal specifications are devoid of such details.

At this point, it is prudent to establish which expressions in a specification are suitable candidates to which visual representations should be attached. The relationship between scenario activities and expressions can be used to establish which expressions correspond to the scenario activities, and which of these may be used as 'hooks' to attach visualisation details. This works as follows. Scenes are packages that define visualisation objects for the actors, AIOs data, and input/outputs. At run time, specification execution generates data values (contained in system state variables). Changes to these variables signify execution progress, and therefore the data in these variables must be visualised. Visualisation is performed by applying the representations contained in the scenes for data, augmented with contextual visualisations. By attaching scenes to the specification a ZAL/ViZ hybrid specification is produced – one that contains the executable model and the appropriate visualisation information which can be processed by the ViZ system at run time to generate the visual prototype.

This analysis does not suggest the form for appearances or suggest how relationships between the elements should be portrayed visually. Instead, this aspect of visualisation remains with the skill of the visualisation designer/developer, in that they must provide the visual cues to form appropriate appearances and visual relationships.

Visualisation Design and Construction - Step 2

A suitable visual metaphor is devised next that characterises the essence of the specification in a form that enables its intent, i.e. its meaning as intended by its authors. An appropriate metaphor should correspond to an underlying mental model the customer might have of the system or of its desired operation. Metaphor design requires the elements identified previously to be analysed in order to reveal their purpose or behaviour in the system. If any represent objects that move physically in the real world, or suggest movement, then suitable animations for these should be designed into the metaphor. The resulting metaphor may consist of a comprehensive design template, encapsulating the rationale for, and dictating the format of any visual representations and animations in the visual prototype. Alternatively, it may simply consist of a general ‘design theme’ that indicates how a visualisation could appear. Metaphor design might be likened to a high-level or conceptual design phase where ideas or themes that are used as the basis for more detailed design are developed.

Visualisation Design and Construction – Step 3

Visualisation implementation, the third step, is performed using the features available in the ViZ toolset that facilitate the construction of appearance and dynamic behaviour of a visual representation, and subsequent storage in the repository. First, the visual representations for the individual components (identified from the analysis step above), such as Abstract Interface Objects, actors, and data elements are created using the facilities afforded by ViZ tool. These correspond to visualisation objects at the intermediate level in the visualisation hierarchy, as described in *Section 2.2.3*. It is possible to acquire images, via an importation mechanism in the toolset, from external software packages or the Internet. This provides greater flexibility and a potentially greater range of images.

Implementation of visualisations can be performed with a view of developing visualisations for use immediately. Alternatively, this can be performed with the intention of developing visualisations for certification, i.e. partitioning visual representations into a class that can be applied, at a later date and by other developers or customers, to a particular software development project. The certification activity is part of the mechanism to assure correspondence between the meaning of a visualisation and the meaning of the specification component to which the visualisation is being applied.

During construction, each component is given a unique identifier by the developer for referencing purposes. In addition, visualisations for execution artefacts are attributed with a type that indicates the type of the execution artefact to which it can be applied. At this point in the process, certification of visual representations can take place, whereby they can be evaluated and categorised by domain experts or developers. Indeed, certification may occur throughout the visualisation implementation stages.

Subsequently, individual visual representations must be combined to form 'expression level' scenes, i.e. visualisations that will portray a complete activity in a scenario. These encapsulate all the intermediate visualisations required for a given expression into a self-contained unit. This visualisation type corresponds to the visualisations at the highest level in the visualisation hierarchy. They are also given a unique identifier by the developer, and are then stored in the repository.

Visualisation Design and Construction - Step 4

The fourth step in this stage is the association of visualisation details to chosen schema in the executable specification. When processed by the ZAL support system, these expressions provide the necessary computational support for generation of values for stimuli, system messages and system state variables. These execution artefacts can be extracted directly from the ZAL system by the ViZ toolset. Therefore, it is necessary to augment the expressions with the visualisations so the ViZ toolset applies the appropriate visual representation at the appropriate time. This entails appending expressions with the references that point to the expression-level visualisation units that were stored previously in the repository.

In addition, it is also necessary to superimpose type information on the system state variables, inputs and outputs in the ZAL specification. Both Z and ZAL can be classified as weakly typed languages, but in order to visualise system state variables, inputs and outputs, it is necessary for the run-time system to determine the type of the data that these represent so it can apply the appropriate visual representation. It is necessary therefore to apply strong typing for this purpose. The mechanics of this entails re-casting data definitions in a different syntax than that of ZAL. At run-time, the ViZ system will interpret this information and use it in the visualisation rendering process.

During this activity, there may be circumstances in which no specification portion can be found that models or supports a particular part of the scenario being considered. If at this stage, such a situation has occurred, it may be said that the requirements specification is incomplete and should thus be reviewed. Again, this provides another opportunity for anomalies to be detected before specification execution, thus facilitating the development of higher quality requirements documents.

3.2.3 Prototype Execution and Evaluation

The final stage in the process involves specification execution and evaluation. There are two steps in this stage: 1) selection of test data, and 2) prototype execution.

Prototype Execution and Evaluation - Step 1

The first step concerns using the scenarios defined in Stage 1 to devise test data, which is used as the basis for a structured prototype evaluation. The scenario descriptions provide details about the inputs required by the prototype. This involves the selection of data that will be used to invoke or exercise a specific scenario. By ensuring that each scenario is exercised, confidence that the requirements are evaluated thoroughly is generated.

Prototype Execution and Evaluation - Step 2

The second step involves executing the prototype using the data defined above. Here, an assessment of the behaviour of the system, as portrayed by the visual prototype is made. This results in dialogue between the developer and the customer about the accuracy of the prototype. If it is found that differences exist in the respective views of the requirements, then negotiation must be undertaken, ultimately resulting in an agreement (or compromise) as to any modifications that are required. Subsequently, this requires changes to be made to the requirements, scenario descriptions, and Z/ZAL specifications. Indeed, re-performing certain aspects of the method, from Stage 1, would be necessary.

The execution of the ViZ process should continue, iterating through several cycles, until a final agreement is reached about the state of the requirements knowledge. The Z specifications that reflect this requirements knowledge would then be used as the basis for full-scale software development.

3.3 The ViZ Toolset

The ViZ toolset provides the practical capabilities necessary to support the execution of ZAL specifications and their subsequent visualisation [Parry00]. The functionality of the toolset is provided by three software components. The first provides visualisation support, the second facilitates the integration of visualisation to ZAL, whilst the third co-ordinates execution and visualisation. This decomposition, and the responsibilities of the components, can be seen in *Figure 3.8.* An elaboration of the respective components is given below. The aim of this elaboration is to build a picture of the whole toolset, enabling its complete architecture, functionality and complexity to be discerned.

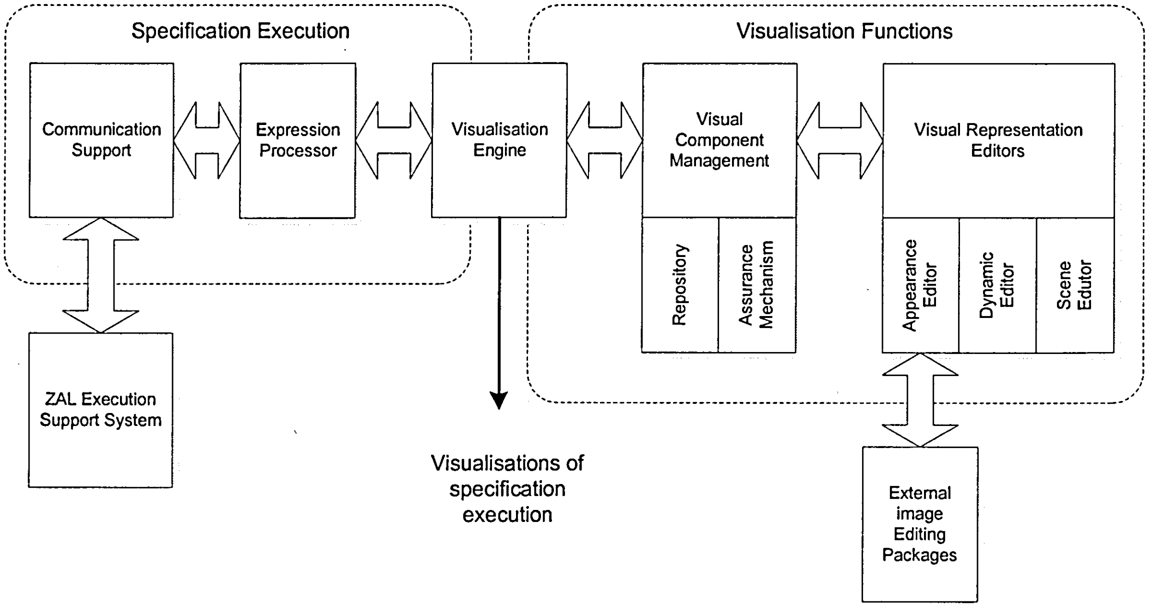


Figure 3.8. The organisation of the functions of the ViZ toolset.

3.3.1 Visualisation Provision

With regard to providing support for visualisation, the toolset offers three important functions. These stem directly from the requirements of the ViZ approach as discussed in *Section 3.1.2*, and are:

1. Facilities for composing and modifying visualisations. Such facilities should accommodate the different levels of visualisation, ranging from the composition of visual cues to form visualisation objects, and the synthesis of visualisation objects into expression-level scenes that can be used to depict the execution of complete expressions.
2. Support for managing visualisations in the form of a visual component repository.
3. A mechanism that assures integrity between the meaning of a visual representation and the meaning of the specification element to which it is being applied.

In terms of functions for visualisation composition and modification, the ViZ toolset possesses three editors [Parry00]: 1) Appearance Editor, 2) Dynamic Editor and 3) Scene Editor (refer to *Figure 3.8*)

The Appearance Editor provides a means of developing the appearance of visualisations from a variety of available visual cues. The Dynamic Editor enables the on-screen location and animation characteristics (i.e. motions) of the appearance to be specified, while the Scene Editor facilitates the synthesis of appearances and motions to form visualisation objects, and further, to develop expression-level scenes.

The primary role of the Appearance Editor is to enable visual cues to be combined to produce a visual representation. The visual cues that are available include text elements and simple geometric shapes. Additionally, the power of third-party graphics packages (such as professional image and photo editing tools) can also be exploited through an importation mechanism whereby images, that are of popular bitmap formats and created with the external software, can be imported and used as visual cues.

The user interface for this editor is based on a mechanism that promotes correspondence between the designer's conceptual view of the desired representation and the actual view while editing. This is achieved by a WYSIWYG style interface that enables visual cues to be specified individually, whilst at the same time giving the creator an opportunity to see the appearance as a whole. In addition, each appearance must be given a unique identifier by the developer.

The appearance definitions for visualisation objects conform to the following structure, described in extended BNF form.

```
<appearance component>      ::=   <appearance component identifier> [<visual-cue>]
<appearance component identifier> ::=  <string>
<visual-cue>     ::=     <cue-> <type> [<attributes>]
<cue>            ::=     <text element> | <shape element> | <image file element>
<attributes>     ::=     <colour> | <image file details> | <text details > | <font details> |
                         <shape details>
<type>           ::=     <integer> | <string> | <set> | <mapping> | <maplet> | <sequence>
```

The Motion Editor is concerned with providing support for the spatial and animation aspects of a visualisation that may or may not have been created. Hence, an appearance and its animation behaviour are regarded as being independent in that motions are polymorphic entities that can be applied to any appearance. This particular editor must support a range of dynamism, stretching from static displays to the graphically animated form. At this static level, the editor can be used to simply define the desired on-screen location which will be associated with an appearance. At run-time, an appearance will be rendered at this given location. To represent dynamic visual forms, the tool allows the path of an appearance to be described in terms of a sequence of nodes, the first node representing the start position for the animation, and the last node representing the resting position. At run time, the appearance will be animated smoothly along this path to depict movement. Again, motions must be attributed with a unique identifier. The structure of these motion components can be described thus,

```
<dynamic component>              ::= <dynamic component identifier>
                                     <start-node> [<node>]
<dynamic component identifier>   ::= <string>
<start-node>                     ::= <node>
<node>                           ::= <horizontal-component> <vertical-component>
<vertical-component>             ::= <integer>
<horizontal-component>           ::= <integer>
```

The design of this editor is based upon a direct-manipulation style user interface. This enables locations to be specified and paths to be described by creating and dragging nodes on a 'canvas' – the canvas representing the screen onto which appearance will be rendered. *Figure 3.9* shows the conceptual design of this interface, and illustrates a typical path with its associated nodes, numbered 1 - 4.

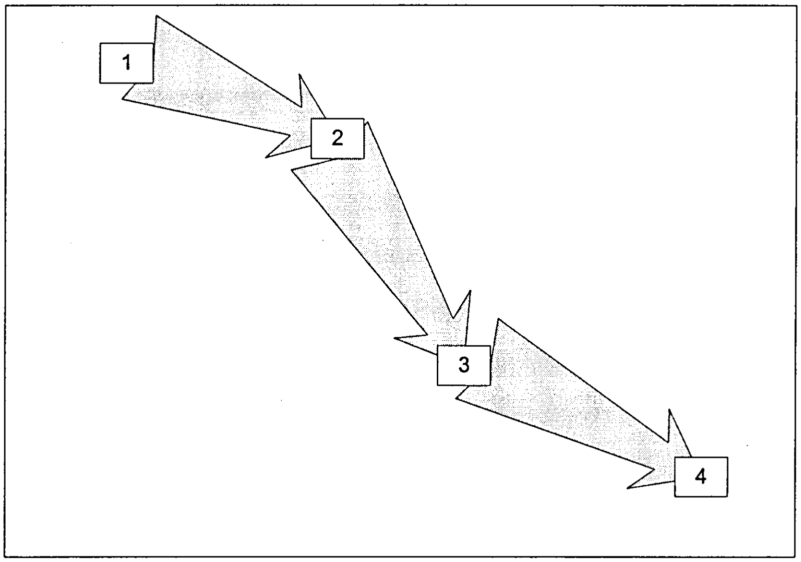


***Figure 3.9**. The conceptual operation of the animation editor.*

The Scene Editor is responsible for enabling individual appearance and dynamic components to be combined to form visualisation objects. It then enables these to be combined to form complete scenes or expression level visualisations. This editor also accommodates the state based nature of expression execution by allowing a developer to specify visualisation objects that should be applied to execution artefacts and rendered before the expression executes, and again after execution has taken place. This is to exploit greater visualisation opportunities, and to enable execution to be presented in a finer level of granularity that provides the viewer with a more real-time view of execution.

This editor possesses a user interface that presents scenes as a hierarchy – a scene consists of before-state visualisations and after-state visualisations, each state consists of visualisation objects, and each visualisation object consists of an appearance and motion component respectively. The editor is such that visualisation objects can be defined for portraying execution artefacts or contextual details that pertain to an expression. For each visualisation artefact to be visualised, a user adds the required number of visualisation objects. Additionally, the user designates these as a particular type – this indicates the type of the execution artefact to which visualisation should be applied, and valid types consisting of integer, set, string, etc.

The structure of scene visualisations is as follows:

```
<scene>                   ::=  <scene identifier>
                               <before visualisations>
                               <after visualisations>
<before visualisations>   ::=  <visualisation object> [<visualisation object>]
<after visualisations>    ::=  <visualisation object> [<visualisation object>]
<visualisation object>    ::=  <appearance component>
                               <dynamic component>
```

Support for managing visual representations is provided through the visual component repository (see Figure 3.8). This acts as a simple database for visual components. It has separate 'containers' in which to store the different categories of components, i.e. appearance, motions, and scenes.

The assurance mechanism is an important aspect of the toolset. This, coupled with the visualisation development activities prescribed in the ViZ process, provides a comprehensive scheme by which the correspondence between the meaning of an execution artefact and the meaning of an associated visualisation can be assured. The mechanism consists of two complementary aspects.

The first aspect is a certification scheme, by which visual representations can be certified as being applicable to a particular software development project. This is facilitated via 'applications containers' in the visual component repository. These can be likened to directories in a file system, whereby components that are related, by virtue of being applied to a particular software development project, can be partitioned.

At design-time, a developer or expert stakeholder can partition pertinent appearance and dynamic components, and scene visualisations, into appropriate containers in order to specify a context in which the components can be used in the future. The editors then place constraints upon which components can be applied – appearance and animation components that belong to a particular application can only be applied to the scenes that also belong to that application. This promotes a certification scheme that can limit the possibility of visual representations being used inappropriately.

The second aspect is a mechanism that limits the application of visual representations to inappropriate types of data. This is achieved by a simple type system. This mechanism brings together the various typing actions performed on the visual components at the early stages in the design process. At design time, visualisation objects, after construction, are attributed with a type to indicate the type of the execution artefact to which the object can be applied. The scene editor contains mechanisms to limit their application to only execution artefacts that are of the same type.

Therefore, using simple and well-understood techniques, such as partitioning and typing, the ViZ system establishes an effective assurance mechanism.

3.3.2 ZAL Integration

To integrate visualisation with ZAL prototype execution, the ViZ toolset requires a number of specific functions. Again, these components are derived directly from the required elements of the system, as elaborated in *Section 3.1.2*. The components are:

1. A means to associate visualisation definitions with specification sections so the toolset can render visualisations at the correct points during execution.
2. An expression processor that is responsible for co-ordinating the execution of individual expressions.
3. A communications interface to facilitate interoperability between the toolset and the ZAL execution support system.

In order for the ViZ toolset to render the chosen visualisations at the correct point during execution, it is required that visualisation details are attached to the specification to indicate which execution artefacts are to be visualised, which visualisation to apply to these, and when. Subsequently, this requires syntax enhancements to the ZAL notation. The enhancements should adhere to the principle of a separation of concerns and provide a minimal connection between visualisation definitions and the underlying specification so not to clutter the specification with extraneous detail. Indeed, this formed one of the criticisms of existing approaches in that some systems freely mixed prototype definition with visualisation details.

With the ViZ system, a concise mapping between visualisation and specification is possible since scene visualisations (which contain all necessary definitions to visualise an expression) are treated as logically separate containers, with their contents treated and stored as such. Scenes, which possess a unique identifier, can then be referred to through this identifier as part of an expression within a ZAL specification. It is then possible for the rendering system to apply in the visualisation objects to the arguments in an expression at run-time. The resulting enhancement to the ZAL notation is therefore,

```
        <expression>        ::=     <operation> <argument set> <scene identifier>
```

This notation is used as the basis for the hybrid ZAL/Visualisation specification that is produced when visualisation details are attached to ZAL specifications.

The details of the syntax enhancements are simply that for each expression that requires visualising, a scene is attached through its identifier. Further, for each argument in the expression, a corresponding visualisation object exists as part of the scene. The order in which the visualisation objects are specified dictate the order in which they are applied to the arguments in the expression – this is performed on a 'first come, first served' basis.

In this design, the onus is placed upon the ViZ tool to apply the visual representations to the arguments in the expression in the sequence specified in the scene visualisation as the execution takes place. This trades syntactic detail in a ZAL specification for functionality in the toolset.

The expression processor is responsible for co-ordinating communication between the ViZ and ZAL software systems. Adherence to the principle of the separation of concerns has resulted in the two separate software systems, each responsible for providing a defined set of functions. This necessitates the two systems to communicate, which in turn requires the two systems to be suitably 'arranged' to facilitate this. The most appropriate arrangement for this application is a client-server architecture, where the ViZ system acts as a client to the ZAL execution support system (the server). Within the chosen operating environment (Microsoft Windows), this is best implemented through the 'Direct Data Exchange' (DDE) facility.

In general, the DDE can be viewed as a mechanism that facilitates communication between applications within the same operating environment. It is based upon a client-server model, whereby a client (seeking a service request) communicates with a server (the service provider). DDE requires 'handshaking' to be enacted so two applications can recognise each other. This is needed since several applications may be sharing data through DDE. To differentiate between applications, a unique 'key' is required to identify the DDE client and server. When a request to communicate occurs, the operating environment polls the DDE interface of each application to see if any will respond to that key. Should this polling result in success then the applications may communicate. Subsequent communication between the applications then depends upon a user-defined protocol.

In the ViZ system, DDE is applied in the following way. First, the expressions processor accepts individual expressions from the visualisation engine. The expressions, represented as simple strings, are communicated through the DDE facility to the ZAL system, and the results are returned, again via the DDE. The protocol that has been defined for the ViZ tool is illustrated in the sequence diagram in *Figure 3.10*. This shows the sequence of messages that are passed between the two software systems as communication is enacted.

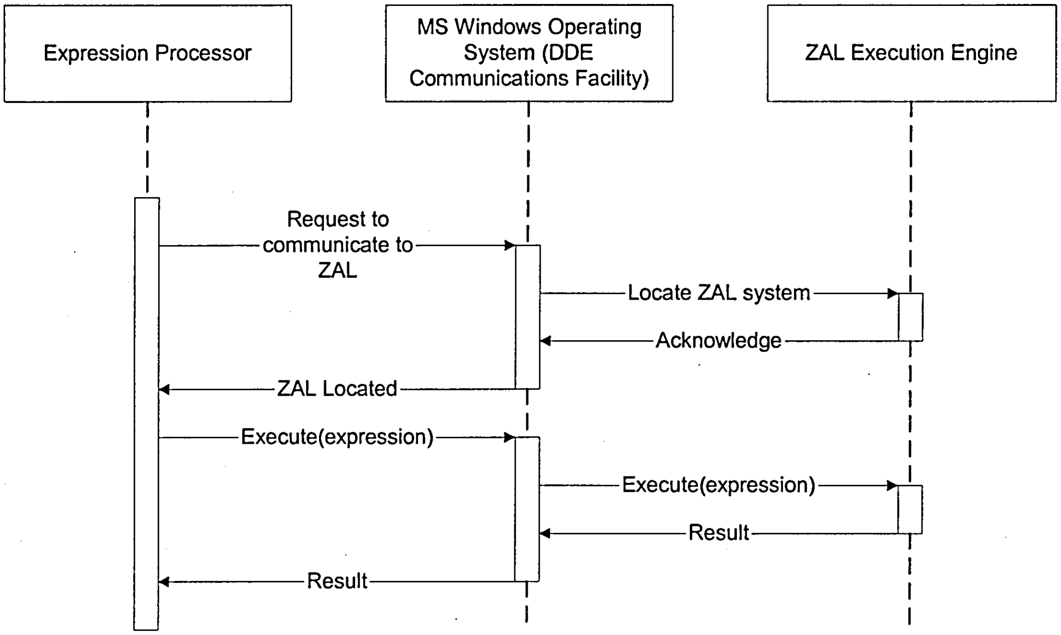


Figure 3.10. *Event sequence characterising the execution of a ZAL expression.*

3.3.3 Visualisation Engine

The Visualisation Engine is the focal point of execution and visualisation activities with regards to the ViZ toolset. It orchestrates schema execution and visualisation through the functionality provided by the other components in the ViZ toolset; it is responsible for the following functions:

1. Co-ordination of ZAL specification execution.
2. Visualisation of execution.
3. Provision of a user interface to enable invocation of the major system functions.

Execution entails processing a ZAL/Visualisation hybrid specification. After a user selects a specification that models the system under consideration, the toolset parses this to identify any syntax errors or unknown visualisation references. A symbol table is also generated consisting of schema names. From this, a list of schemas is presented to the user. Execution commences when the user selects a schema that represents the system functions to be validated.

Execution, and subsequent visualisation, involves sequential decomposition of each expression in the selected schema, whereby the ZAL component and the scene reference is separated. At this point, the 'before-stage' of the visualisation is rendered. This is achieved by interrogating ZAL as to the values of the system state variable referred to by the arguments in the expression. Rendering is performed in accordance with the visualisation objects contained in the 'before-stage' section of the scene. The contents of the visualisation objects are retrieved from the ViZ repository. After rendering, the whole expression is passed to ZAL, via the expression processor for execution, and the system state, held by ZAL, is updated. The visualisation engine also collects any specific results that are returned from this execution. Lastly, the 'after-stage' visualisations are applied. Again, this is performed by first interrogating ZAL as to the contents of the arguments involved as well as any results, and second applying the visualisation objects in the 'after-stage' section of the scene. This course of action is continued until no more expressions remain.

In terms of the user interface, facilities are provided for a user to invoke execution. Controls are also offered to pause and resume execution when it is underway. In addition, the user interface provides a means to access the editor and repository functions. To this end, this component serves as the primary interface between the system capabilities and the user.

3.4 Summary

This chapter has presented a description of an alternative prototype visualisation system, known as ViZ. The aim of the system is to facilitate user validation by providing capabilities to visualise the execution of prototypes developing using a formal methods-based prototyping approach. The foundation for the system is rooted in overcoming shortcomings that were identified with existing systems, and incorporating features that are deemed desirable to provide effective visualisation. This chapter described the characteristics of the two facets of the ViZ system, namely the software tool and the associated methodology.

The following chapter extends this description by presenting a series of case studies that illustrate how the ViZ approach can be used within the context of user validation. These case studies highlight the possibilities and potential of the approach, and are based upon specific examples of the invocation of the ViZ method and usage of the toolset.

Chapter 4
ViZ System Validation and Case Studies

In the previous chapter, an enhanced prototype behaviour visual system, ViZ, was introduced and described. The system possessed characteristics that enable the execution of prototypes developed with the formal-methods based RealiZe approach to be visualised. This chapter continues by describing the application of the ViZ system to four separate case studies. The aim is to provide a validation of the system by demonstrating both its capabilities and the effectiveness of visualisation in the context of requirements validation. Each case study invokes the ViZ process and toolset for a different specification. The case studies that will be considered are:

1. An *Automatic Teller Machine (ATM)* system.
2. An *email system.*
3. A safety critical *Water Level Monitoring System (WLMS).*
4. A *security system.*

The chapter is divided into four sections, each documenting a particular case study. Each case study is presented in the style of a scientific experiment that describes five aspects, namely *aims*, *context*, *method*, *results*, and *observations and conclusion.* The aims state the overall objectives of the given case study. The context describes the requirements and the specification that is being used to demonstrate those aims. The method presents a narrative detailing the application of the ViZ process to produce a visual prototype. Results present the outcome, i.e. the resulting visualisations of prototype behaviour during execution. Lastly, observations and conclusions offer a discussion centred on the effectiveness of the demonstration and in particular how the results relate to the objectives.

4.1 Case Study – An Automatic Teller Machine

This case study presents the development of a prototype of the behaviour of an Automatic Teller Machine (ATM) that is typically found in the domain of retail banking. The prototype would then be used as a vehicle to stimulate discussion and evaluation in the context of requirements validation.

4.1.1 Aims

The aims of the ATM case study are threefold:

- To show, primarily, the ViZ system achieving its objective in extending and enhancing the presentation of ZAL- prototype execution.
- To demonstrate the application of the ViZ method as an effective process for detailing the steps inherent in developing a visual prototype.
- To illustrate the general capabilities of the ViZ system and the utility of the process and toolset, and especially to show the different types and styles of visual representations that the system can accommodate.

The presentation of this case study is comprehensive in nature; for instance, the activities inherent in visual prototype development are elaborated in full. This is to facilitate the demonstration of the utility and power of the ViZ system.

4.1.2 Context

The requirements of the ATM system are based upon those found in [Regnell95]; they relate to the control of a typical stand-alone ATM machine, and are as follows:

- The system will store information relating to one of more customers, such as account number, PIN number, and the balance of each account.
- The system will update customer records and the amount of money in the machine currently available for withdrawal.
- Customers possessing a card, encoded with their account number, will be recognised by the system.

The ATM system will allow cash to be withdrawn from the machine. To do this, the customer will be required to insert a card into the machine from which their account number is read. The customer shall then be requested to enter their PIN via a keypad on the machine. This is validated against the PIN held for that particular account. Upon successful PIN validation, the customer is then requested to enter the amount of money they wish to withdraw, again via the keypad. If the transaction is valid, i.e. the amount of money requested does not exceed the balance of the account and the machine has sufficient funds to cover the request, then the machine shall eject the requested amount through the cash dispenser. Finally, the machine will eject the customer's card.

In the event of an unsuccessful withdrawal, the machine will issue a suitable error message, depending on the condition.

The hardware of the ATM machine comprises of a card reader, into which the customer's card is inserted, a message responder, i.e. a display screen, a keypad, and a cash dispenser [Regnell95].

Given these informal requirements, a Z specification that serves to model them can be developed, thus:

```
[PIN, NAME, AMOUNT, MESSAGE, ACCOUNT]

┌ATM System ─────────────────────────────────────
│ name          : ACCOUNT ⇸ NAME
│ pin           : ACCOUNT ⇸ PIN
│ balance       : ACCOUNT ⇸ AMOUNT
│ cards         : P ACCOUNT
│ available     : AMOUNT
└──────────────────────────────────────────────

┌Withdraw Money ─────────────────────────────────
│ ΔATM System
│ request?            : AMOUNT
│ card?               : ACCOUNT
│ pin?                : PIN
│ request?            : AMOUNT
│ money_message!      : MESSAGE
├──────────────────────────────────────
│ card? ∈ cards
│ pin? = pin(card?)
│ request ≤ balance(card?)
│ request ≤ available
│ balance' = balance ⊕ {card? ↦ (balance(card?) - request?)}
│ available' = available - request?
│ money_message! = `Take_your_money
└──────────────────────────────────────────────
```

From this specification, and using the TranZit tool, a corresponding ZAL specification can be derived. This needs to be augmented with a system state definition and instantiated with appropriate sample data, in this case representing typical card details, account balances, etc. The resulting transformed executable specification, shown below, provides a starting point from which the ViZ process can be applied.

```
(make cards { 101 102 103} )
(make name [ #(101 Stan) #(102 Eddie) #(103 Hilda) ])
(make balance [ #(101 1000) #(102 50) #(103 250) ])
(make pin [ #(101 5671) #(102 8819) #(103 2350) ])
(make available 550)
(schema WithdrawMoney
    (? card?
     ? (pin? request?)
     ! money_message!
    )
    (and
         (mem card? cards)
         (equalp pin? (applyz pin card?))
         (lessorequal request? (applyz balance card?))
         (lessorequal request? available)
         (make temp (- (applyz balance card?) request?))
         (make balance' (override balance [#(card? temp)]))
         (make available' (- available request?)
         (make money_message! 'Take_your_money)
    )
)
```

4.1.3 Method

Application of the ViZ process entails the undertaking of the three stages with their inherent steps, i.e.:

Stage 1 - Scenario Identification and Documentation

Step 1 Determine system functions of interest

Step 2 Identify and document scenario descriptions

Stage 2 - Visualisation Design and Construction

Step 3 Analyse scenarios to establish visualisation targets

Step 4 Develop a suitable visual metaphor/scheme for the whole visualisation

Step 5 Construct visual representations using the ViZ tools

Step 6 Apply visualisations to the ZAL specification to produce the visual prototype

Stage 3 - Prototype Execution and Evaluation

Step 7 Devise test cases from scenarios to guide the prototype evaluation

Step 8 Execute and evaluate visual prototype

Scenario Identification and Documentation

The first step in identifying and documenting scenarios entails establishing which system operations are to be validated (and hence visualised). In terms of the ATM specification, the function for money withdrawal will be under scrutiny.

Identify and Document Scenario Descriptions

The second step involves development of the scenario descriptions. This is achieved by selecting scenarios that are of interest from the possible set of scenarios that could pertain to the system function being validated. In practice, this can be achieved through identifying the significant executable pathways through the executable specification.

With regard to the potential scenarios for the money withdrawal function, it can be seen from the informal requirements and the executable specifications that two outcomes become apparent: successful and unsuccessful money withdrawal. Success can be characterised as:

$$\text{success} \Rightarrow (\text{card accepted} \wedge \text{pin accepted} \wedge \text{enough money in machine} \wedge \text{balance adequate})$$

Failure can occur when any one of four conditions is not met:

$$\text{failure} \Rightarrow (\text{card not accepted} \vee \text{pin not accepted} \vee \text{not enough money} \vee \text{customer balance inadequate})$$

From these, five possible scenarios can be identified: a single scenario with the objective of describing the activities inherent in successful withdrawal of money, and four others that describe the failure to satisfy any of the four conditions. The ones that are of interest are selected for furnishing with specific details.

For the purpose of this case study, the derivation of the 'Successful Money Withdrawal' scenario will be described in detail. The scenario description is developed as follows. First, a general structure in terms of the system behaviour the scenario should represent is established directly from the informal requirements. From the specification, the behaviour of the system can be seen as:

1. Validate card
2. Validate PIN
3. Validate amount required
4. If the card, PIN, and amount are valid, then eject card, then dispense the money

The Successful Withdrawal scenario is a specialisation of this behaviour. It requires details that pertain to the specifics of successfully validating the card, PIN and the amount, and demonstrating the successful dispensation of money and the card. In addition, these key operations require contextualising if they are to be presented in a user-centred form. This entails adding extra details that characterise how a user would actually interact with the system and how and when the system would present results and responses. The Successful Withdrawal scenario requires activities that describe how the user enters the card, PIN and amount details into the machine. These refinements are:

1. Customer inserts card into the card reader
2. Card is validated
3. Customer enters PIN
5. PIN is validated
6. Customer enters required amount of money
7. Validate amount
8. Eject card
9. Dispense money

Subsequently, a detailed scenario description, based on the above list can be derived that contains not only activities, but also actors, Abstract Interface Objects, and data/messages, and inputs/outputs. Such a detailed scenario description specifies the relationships between the activities, actors, AIOs and data. This refinement is performed by taking each activity in turn and establishing the input stimuli it requires (from actors), and any outputs that are generated (from the system). The finalised form of the 'Successful Withdrawal' scenario is shown in the sequence diagram in *Figure 4.1*, which uses the notation described by Regnell [Regnell95]. This shows the actors (the source of input data), the AIO that accepts the data and passes in to the system operations, and the outputs/messages that result. The activities that comprise this scenario are those with which the user/actor has direct interaction, and internal system actions such as updating system state variables are not featured. It should be noted that to develop more comprehensive visual prototypes for larger systems, establishing the details of exceptional cases, to deal with unrecognised or erroneous inputs for example, would also have to be performed at this point.

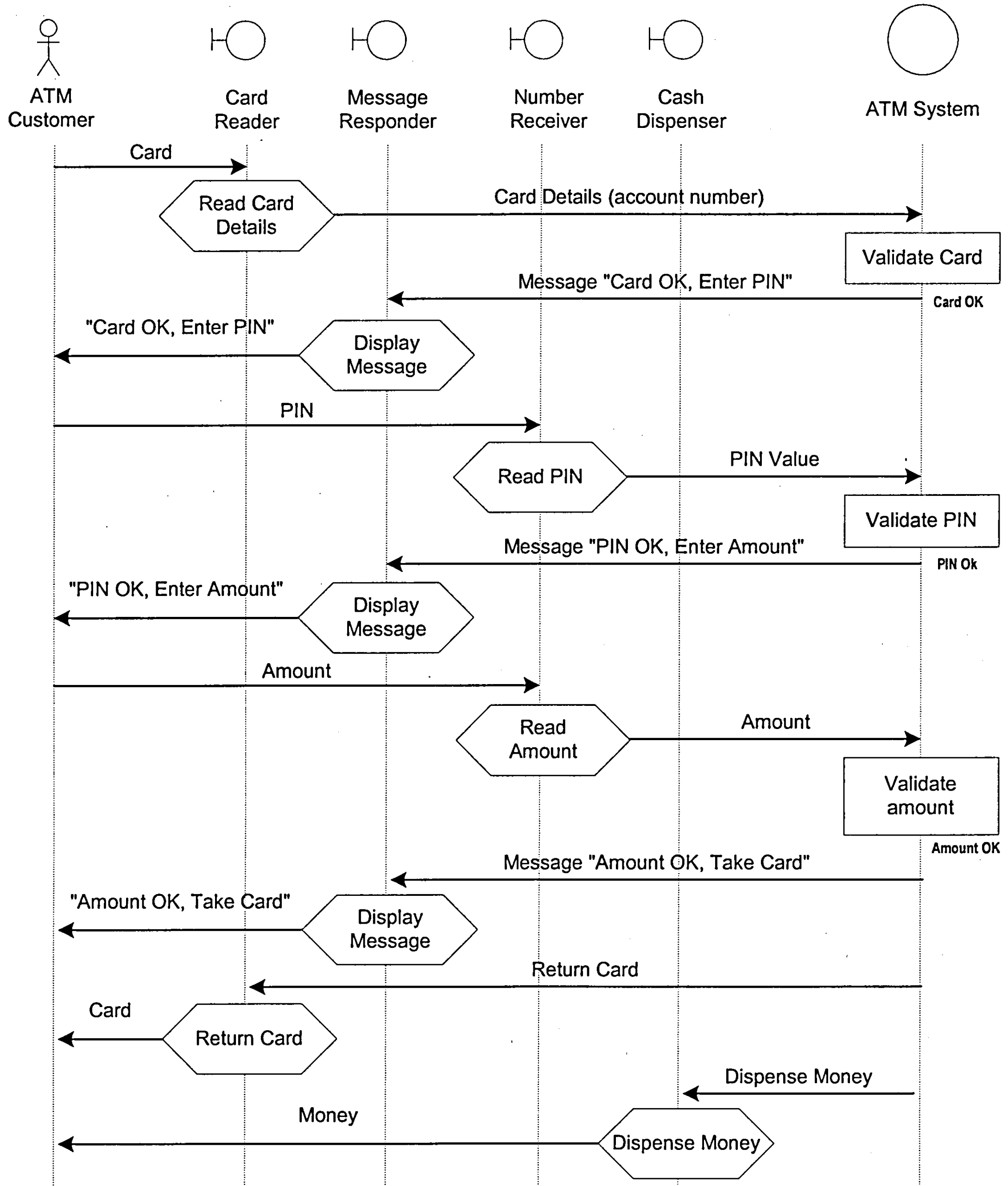


Figure 4.1. Sequence diagram showing the scenario of the successful withdrawal of money.

Visualisation Design and Construction

There are four steps in developing visualisations: scenario analysis to identify structure and form of a potential visualisation, visual metaphor design, construction of visual representations using the ViZ tools, and their subsequent association with the specification (see *Section 3.2.2*). This case study illustrates the application of these steps but does not however show certification of visual representations prior to their use – this will be demonstrated in *Case Study 3*.

The major consideration during this stage is to devise visualisations that characterise the execution of the prototype using forms that are suggestive of the meaning of the underlying requirements, and encapsulating them in a storyboard fashion. Visualisation design is not performed through software automation, but instead relies upon the developer to interpret requirements, executable specifications and scenario descriptions, and to exploit the opportunities for visualisation that are suggested as the design steps are undertaken. This contrasts to the steps in constructing visual representations, where software support, through the features in the ViZ tool, is employed throughout.

Scenario Analysis

Scenario analysis (with the aim of determining the potential structure of the visualisation) is achieved by simply devising notional scenes for each scenario activity that will enable that activity to be portrayed visually. This is achieved by exploiting the relationship between scenarios and visualisation that indicates that one scene is required per scenario activity.

Table 4.1 shows one possible visualisation structure, in terms of scenes, that can be established for the chosen scenario. This table is composed from: the scenario descriptions originally identified, possible scenes identified from the sequence diagram (*Figure 4.1*), and the corresponding ZAL related expression. Identifying these expressions provides an indication as to the points in a schema to which the scene definitions should be attached, so, at run time, the ViZ system can process the specification and appropriately apply the visualisations in the associated scenes. The actual attachment is performed later when the scenes have been defined using the ViZ tool.

Scenario Activity	Possible Scenes	Related Specification Expression
1. Customer inserts card into the card reader 2. Card is validated	Scene 1. Show customer inserting card into machine. (used to show the state of the system before executing the expression). Scene 2. Show results of card validation (used to portray the state of the system after the expression is executed).	(equalp pin? (applyz pin card?))
3. Customer enters PIN 4. PIN is validated	Scene 3. Show customer entering PIN on keypad. (used to show the state of the system before executing the expression). Scene 4. Show results of PIN validation. (used to portray the state of the system after the expression is executed).	(lessorequal request? (applyz balance card?))
5. Customer enters required amount of money 6. Validate amount	Scene 5. Show customer entering requested amount using keypad. (used to show the state of the system before executing the expression). Scene 6. Show results of amount validation. (used to portray the state of the system after the expression is executed).	*(lessorequal request? (applyz balance card?))* (lessorequal request? available)
7. Eject card	Scene 7. Show card being ejected	
8. Dispense money	Scene 8. Show money being dispensed. (used to portray the state of the system after the expression is executed).	(make money_message! 'Take_your_money)

Table 4.1. *Correspondence between scenarios and expressions, and possible scenes required to visualise these.*

The scenes identified facilitate a user centred portrayal of the execution of the specification. This is achieved by concentrating the visualisation effort on the activities that a user would be directly involved with or would otherwise experience. The visualisation structure 'leans' in this direction in that only the activities that are relevant to a user are earmarked for visualisation. This is reflected in the lack of scenes to depict the typical housekeeping activities that update internal system state variables. A typical ATM user (i.e. a bank customer) would not necessarily be interested in these details – they would instead be more interested in collecting the money they have requested. This is reflected in the presentation of reassurances, through system messages, on the status and progress of the money withdrawal. However, an issue arises as to the definition of user. Although a user of the ATM machine may not be interested in internal housekeeping details at prototype level, the bank, which may be regarded as a customer, insofar as they might the responsible for procuring and financing the system, might show a keen interest – they want reassurance that the system does indeed behave as they require, and therefore would like to see how the system state is updated when transactions are processed.

It can be seen in *Table 4.1* that for most scenario activities, one or more schema expressions are used to model them. However, there is one exception, namely the activity '7. Eject Card'. Here, no expression can be identified that models this. Such situations may be symptomatic of an inaccurate specification, i.e. the specification omits a particular step and is erroneous. Thus, by performing this type analysis, such as identifying correspondences between scenarios and executable specifications, errors in specifications (or errors in scenario descriptions) can be identified and corrected early. In this particular instance, and after careful inspection, it is decided to amend the specification, and the corresponding ZAL form, thus:

```
┌Withdraw Money──────────────────────────────────
│ ΔATM System
│ request?              : AMOUNT
│ card?                 : ACCOUNT
│ pin?                  : PIN
│ request?              : AMOUNT
│ card_message!               : MESSAGE
│ money_message!        : MESSAGE
├──────────────────────────────────────
│ card? ∈ cards
│ pin? = pin(card?)
│ request ≤ balance(card?)
│ request ≤ available
│ balance' = balance ⊕ {card? ↦ (balance(card?) - request?)}
│ available' = available - request?
│ card_message! = `Take_your_card
│ money_message! = `Take_your_money
└──────────────────────────────────────────────
```

```
(schema WithdrawMoney
    (? card?
     ? (pin? request?)
     ! (card_message! money_message!)
    )
    (and
         (mem card? cards)
         (equalp pin? (applyz pin card?))
         (lessorequal request? (applyz balance card?))
         (lessorequal request? available)
         (make temp (- (applyz balance card?) request?))
         (make balance' (override balance [#(card? temp)]))
         (make available' (- available request?)
         (make card_message! 'Take_your_card)
         (make money_message! 'Take_your_money)
    )
)
```

To establish the indicative content of the scenes, identification of targets for visualisation is performed. Scenes should contain a visual representation for each important element found in the scenario activities. Therefore, for each actor, Abstract Interface Objects (which form the interfaces between the system and the actors), and data elements, a corresponding visual representation is required. By determining which actors, AIOs and data elements feature in the scenario activities, it is possible to establish how many visual representations are required. These elements can be established directly from the scenario descriptions. Once identified, a visualisation object, comprising an appearance and dynamic component, can be developed for each of these elements. When developed, the visualisation objects that pertain to scenario activities are subsequently packaged together to form complete scene definitions, which will in turn be attached to the specification at the point identified in the scenario analysis step.

Taking the first activity in the chosen scenario as an example (i.e. "1. Customer inserts card into the card reader"), it can be seen from *Figure 4.1* and *Table 4.1* that the actor is the 'ATM Customer', the Abstract Interface Object is the card reader, and the data is the card identifier (an input to the system, that represents the user's ATM card). The complete set of elements that can be identified for all the activities in the Successful Withdraw Money scenario can be seen in *Table 4.2*.

Successful money withdrawal		
Actors	**AIOs**	**Data**
1. Customer inserts card into the card reader		
ATM Customer	Card reader	Card (input value)
2. Card is validated		
ATM Customer	Message responder	"Card OK" (message)
3. Customer enters PIN		
ATM Customer	Number receiver/keypad	PIN (input value)
4. PIN is validated		
ATM Customer	Message responder	"PIN OK" (message)
5. Customer enters required amount of money		
ATM Customer	Number receiver/keypad	Amount (input value)
6. Validate amount		
ATM Customer	Message responder	"Amount OK" (Message)
7. Eject card		
ATM Customer	Card reader	Card (Message)
8. Dispense money		
ATM Customer	Cash dispenser	Cash amount (Message)

Table 4.2. *ATM Scenario activities and important elements that can be visualised.*

Once the information has been gathered with regard to the possible structure and content of the visualisations, the next activity is to devise a visual metaphor or visual scheme upon which the format and style of the visualisations will be based. This metaphor should offer the viewer recognisable representations of physical artefacts that correspond to their own perceptions and mental models of the ATM system and its operation. At this point, it might be prudent to explore the possibility of using established icons or representations that provide an already agreed upon or immediately recognisable visual form. Such established representations may exist already in the domain, or have been developed previously by domain experts, or developed as part of research efforts along these directions [Spence01]. However, for the purpose of this case study, no such established representations will be sought or used, since its purpose is to demonstrate their development and application.

For the ATM System, a suitable metaphor would be that of a realistic presentation of an actual ATM System, including realistic images for the actor and abstract interface objects, as well as the actor stimuli and system responses.

The associated design of the visualisations for the scenarios, using this metaphor, can be seen in *Figure 4.2*. This shows, in the manner of a sketched storyboard, the envisaged format of the visualisations and associated animations and the obvious influence of the visual metaphor. In this design, it is necessary to ensure that the frames in the storyboard correspond to the scenes identified in the previous analysis stages, and the pertinent aspects, such as actors, data and abstract interface objects feature in the scenes.

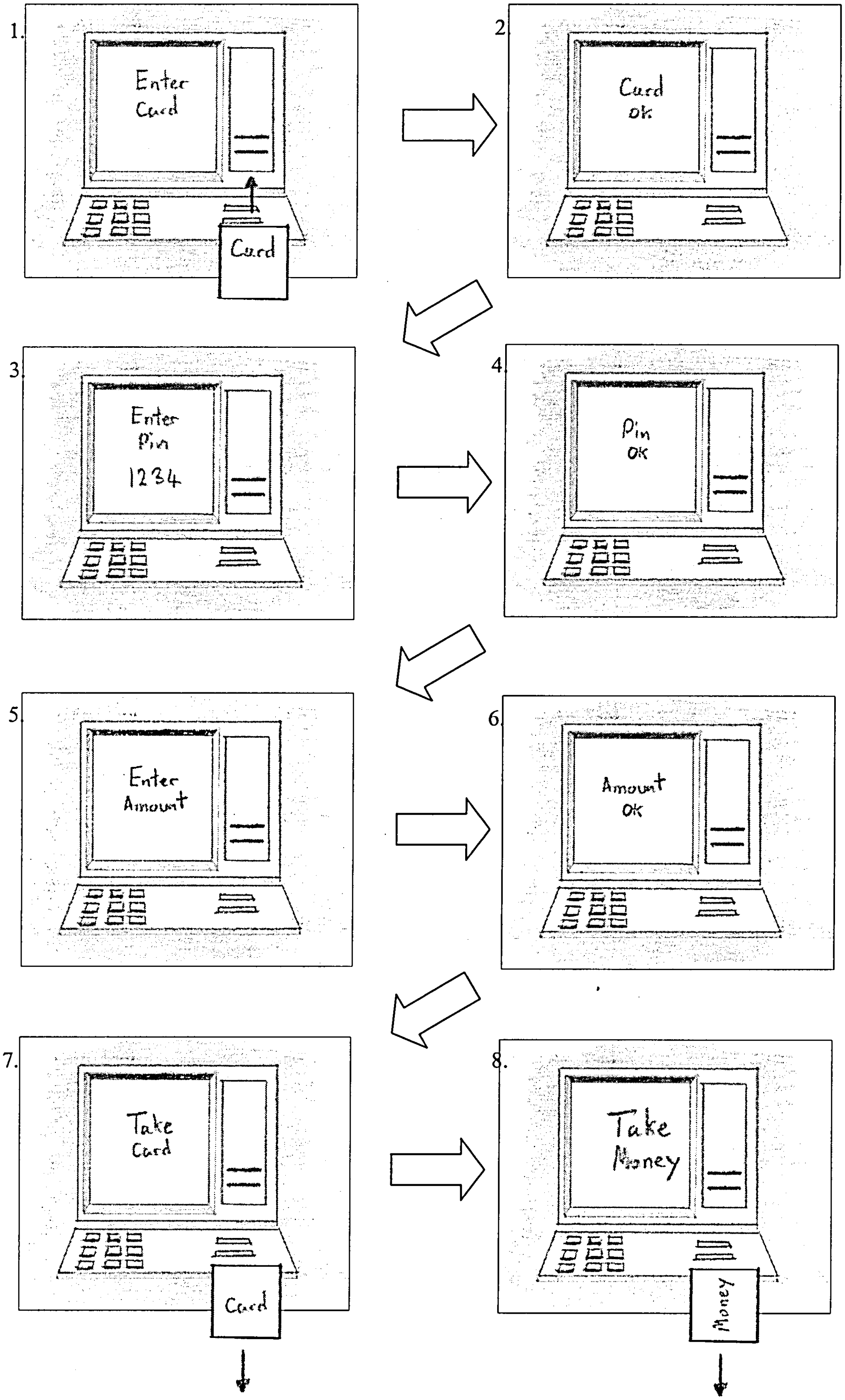


Figure 4.2. *Conceptual design of the visualisation for the Successful Money Withdrawal scenario.*

Visualisation Construction

Visualisation construction follows design. In practice, construction involves four sub-activities:

1. Initialisation of the ViZ repository to create suitable storage containers for visual representations that are developed,
2. Construction of individual visualisation objects to represent the scenario elements identified above,
3. Creation of expression level scenes for each scenario activity (also identified above),
4. Attachment of these scene definitions to the ZAL specification. To demonstrate these activities the development of a complete scene will be presented.

The scene chosen for this purpose represents the activity '*1. Customer inserts card into the card reader*'.

First a suitable container in the visual component library is created. In this case, an 'ATM application' container is created in the library, as shown in *Figure 4.3*.

Next, construction of visualisation objects entails the use of the editors in the ViZ toolset to produce concrete visual representations based on the above design. The order of development is appearance of visualisation objects first and screen position/animation details second.

For each element in the chosen scenario, a suitable appearance is devised. For this case study, the design brief states that realistic images should be used, so it is therefore necessary to obtain suitable images of the ATM entities. *Table 4.3* shows the images selected to represent the elements in the scenes used to portray the selected scenario activity.

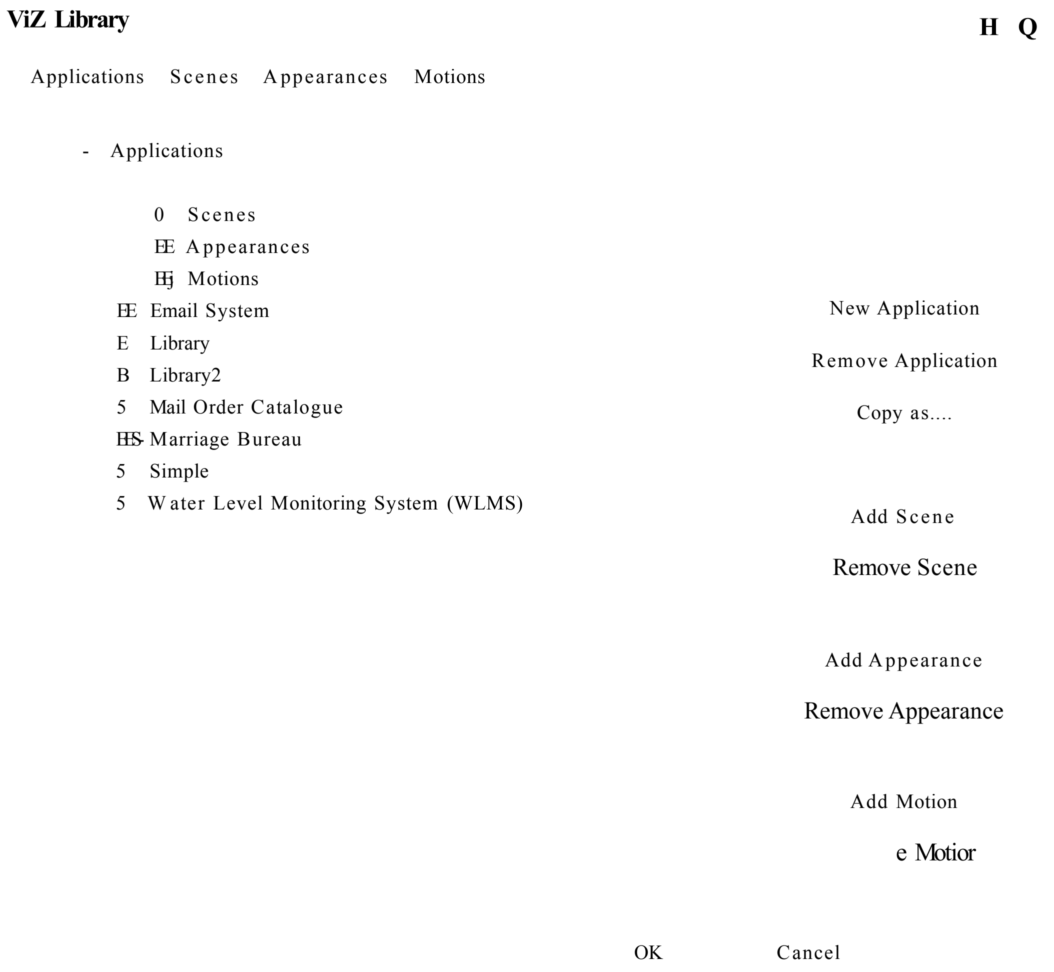


Figure 4.3. The ATM application container in the visual component repository.

The issue of selecting images that are suggestive of the meaning of the entity that they are supposed to represent again becomes prominent. In this case study, certification of representation prior to their use is not considered, and therefore retaining integrity between the meanings of visual representation and specification is the responsibility of the developer/visualisation designer. However, due to the analysis and decomposition of the requirements, potential visualisations and scenarios that are offered in the preceding stages of the ViZ process, the developer is armed with information they can use when constructing appropriate imagery. For the ATM system, which has readily identifiable entities (such as actors, the ATM machine, cards, etc.), constructing suitable representations may be regarded as being a straightforward activity. For complex systems, there may be subtle differences between entities, or a lack of understanding of their role within the system or their meaning in a certain context. This may present difficulties. The analysis stages however, may provide the developer with an opportunity to understand the system in more detail, so they can develop appropriate representations without having to resort to employing approximations or educated guesses.

	Element	**Image**
	ATM Customer	
Scene 1	Card reader	
	Card (input value)	
Scene 2	ATM Customer	
	Message Responder	

“Card OK” Message	*t S g m* ■

Table 4.3. The images to represent the entities in the scene “Customer inserts card into card reader.”

Using the ViZ Appearance Editor, and for each visualisation object, the images are imported into the ViZ system, and attributed with a unique identifier. *Figure 4.4* shows the card image in the Appearance Editor. Where appropriate, the visualisation objects are adorned with other attributes. One such important attribute is a flag that indicates to the ViZ system to display the contents of a system state variable, result, or other execution artefact alongside the image. This not only enables values or execution results to be presented, but also enables the data values be associated with a particular image, strengthening the understandability of the value or result. This is achieved by specifying, through the separate ‘Attribute Panel’ in the Appearance Editor, where the value should be displayed in relation to the image. In addition, if the visualisation object is to be used to portray execution artefacts, then they must be attributed with a type - the data type of the system state variable to which the object will eventually be applied. After creation, all visualisation objects are stored in the visual component repository, in the project container that was created for the ATM Application.

Appearance Editor S

Bitmap Resources

Bitmap List: Preview:

C:\S nappy32\l MAGES\atmcard. bmp

CE3 £

Add. Help

Change.... Cancel

Remove.. OK

a-

Attribute Editor r n

Appearance Name: |ATM Card

Background Frame Borders

W Enable background ⊦ Enable frame Top: [tT

Colour Colour Bottom: fo"

Left: [O'

Change Colour

Right: [ci-

Options Data Attributes

r Single flag Appearance: |ElementApp

⊦ Display flag

Range Appearance: |"

Across Position : X |l 00 Y [205~

C' Down

Separator: fo"

Font: |Arial Narrow Change Font

Text Colour: Background Colour:

Cancel

Change Colour Change Colour

OK

Figure 4.4. Appearance editor, (above) shows the Appearance Editor with the imported card image and associatedparameters, (below) shows the Attribute Panel that is used to specify options, data types, and additionalparameters.

After the appearances have been created for the visualisation objects, the next step is to specify the screen positions for the visualisation objects and specify any animation paths that may be required in the scene - the paths indicating the route that an appearance will travel when the visualisation is shown.

Taking the design of the scene as a basis, the Motion Editor is used to construct separate spatial components for each visualisation object in the scene. The Motion Editor enables screen locations to be specified in an interactive manner, as shown in *Figure 4.5.* The grid is used as a reference system, from which positions can be noted and discussed without having to resort to a low-level pixel based co-ordinate scheme.

S3 Edit Motion Obfect

Motion

Figure 4.5. Demonstration ofMotion Editor being used to specify the on-screen locationfor a visualisation object.

For the scene that portrays the scenario activity being considered, one animation is required. This is for an animation of the ATM card sliding into the machine to depict the actions of a customer inserting a card into the card reader (refer to the conceptual visualisation design shown in *Figure 4.2).* The Motion Editor can be used to specify this motion by indicating a start and end point that an associated visualisation object will follow during execution. *Figure 4.5* shows the Motion Editor being used to describe a simple path for the Card visualisation object. At run time, and when associated with the Card visualisation object, the card image will animate smoothly along the path, suggesting the card moving into the machine

Each spatial/animation component is also attributed with a unique identifier and stored in the ATM Application container in the visual component repository. At this point both appearance and motion components are separate entities. It is only at a later stage that they are coupled to form complete visualisation objects for a particular entity. This separation facilitates a degree of polymorphism and hence reuse, where a description of a motion may be applied to one or more appearances.

After the individual appearances and motion components have been constructed, the next step is to combine these to create visualisation objects to represent execution artefacts or contextual details. This is done in the context of the scene currently under development, and is achieved using the ViZ Scene Editor. The Scene Editor enables a developer to specify visualisation objects that should be applied before an expression is executed and again afterwards. The definition of the scene used to depict the execution of the chosen scenario activity can be seen in *Figure 4.6*. This shows the completed expression level visualisation, presented in a tree-like manner, comprising two main sections. The first describes the 'before' visualisation objects (corresponding to Scene 1), whilst the second describes the 'after' visualisations (corresponding to Scene 2).

Scene visualisations are attributed with a unique identifier by the developer and are then stored in the repository for later use. Scene visualisations are developed for each of the scenes identified in earlier in the analysis phase (see *Table 4.1*). For this particular scenario, the names of the scene are shown alongside their respective scenario activities and the expressions to which they will be attached in *Table 4.4*.

The structure of scene visualisations is such that they specify the visualisation objects to be applied to the execution artefacts. At run-time, the ViZ system applies these to the execution artefacts in the expressions to which the scene visualisations are attached on a sequential basis.

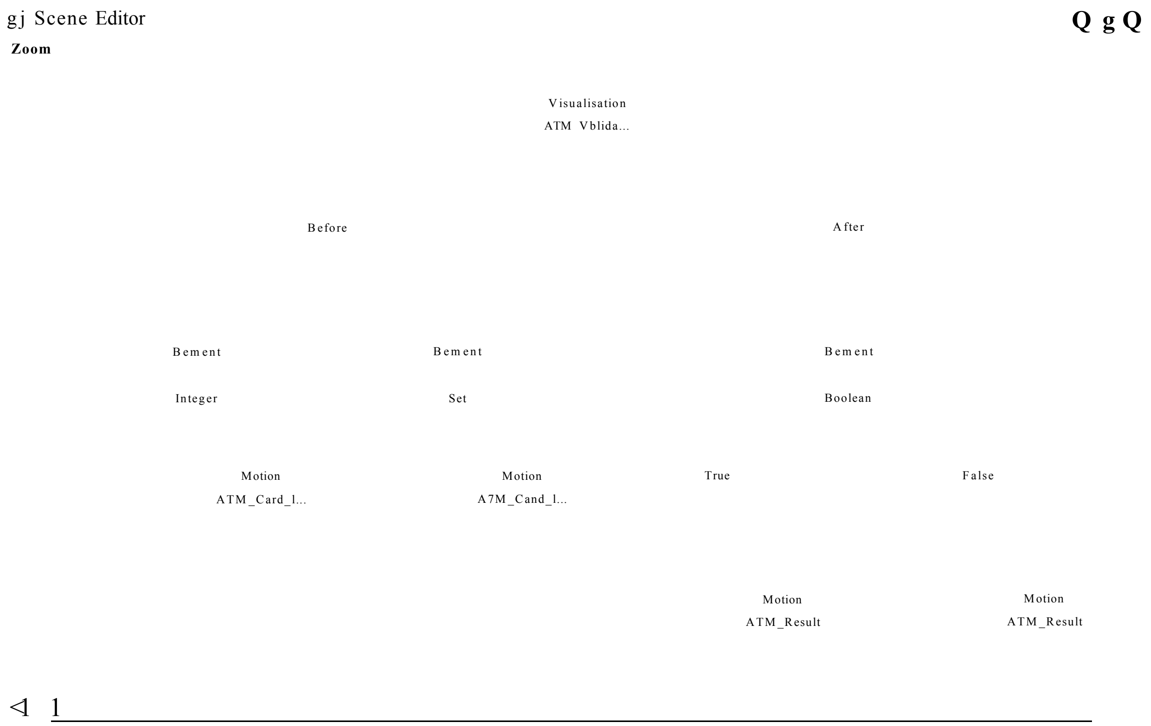


Figure 4.6. The completed scene used to represent the first activity in the chosen scenario.

Scenario activity	Related expression	Scene Identifier
1. Customer inserts card into the card reader 2. Card is validated	(equalp pin? (applyz pin card?))	ATM_ValidateCard
3. Customer enters PIN 4. PIN is validated	(lessorequal request? (applyz balance card?))	ATMJValidatePIN
5. Customer enters required amount of money 6. Validate amount	*(lessorequal request? (applyz balance card?))* (lessorequal request? available)	ATM_ValidateBalance
7. Eject card	(make card_message! 'Take_your_card)	ATM_EjectCard
8. Dispense money	(make money_message! Take_your_money)	ATM_DispenseMoney

Table 4.4. Correspondence between scenarios, expressions, and the names of the scenes that contain the visual representations necessary to visualise them.

Apply Visualisations to Specification

Following visualisation design and construction, the final step is to associate the completed scene visualisations with the corresponding expressions.

Firstly, it is necessary to superimpose type information on the system state variables, inputs and outputs in the ZAL specification. This is a straightforward activity. Next, the expressions that model the scenario activities are appended with the unique identifiers of the relevant scenes. The results are reflected in the ViZ/ZAL hybrid specification below:

```
(data set cards {101 102 103})
(data mapping pin [ #(101 5671) #(102 8819) #(103 2350) ])
(data mapping name [ #(101 'Stan) #(102 'Eddie) #(103 'Hilda) ])
(data mapping balance [#(101 1000) #(102 50) #(103 250) ])
(data integer available 550)

(schema WithdrawMoney
    (header   input card?
              input pin?
              input request?
              local card_message!
              local money_message!
              local available'
              local balance'
              local temp
    )
    (and
         (mem card? cards ATM_ValidateCard)
         (equalp pin? (applyz pin card?) ATM_ValidatePin)
         (lessorequal request? (applyz balance card?))
         (lessorequal request? available)
         (make temp (- (applyz balance card?) request?))
         (make balance' (override balance [#(card? temp)]))
         (make available' (- available request?) )
         (make card_message! 'Take_your_card ATM_EjectCard)
         (show card? ATM_TakeCard)
         (make money_message! 'Take_your_money)
         (show request? ATM_TakeMoney)
    )
)
```

Prototype Execution and Evaluation

Effective evaluation of the prototype is the purpose of the whole ViZ methodology. It enables a dialogue between the developer and stakeholders to take place that facilitates discussion and negotiation about the requirements. Evaluation takes places by executing the prototype.

The first step in execution is to prepare test data that will enable the prototype to be exercised sufficiently to present the execution of the selected scenarios. Once again, one returns to the scenario descriptions. Each scenario will require appropriate data inputs that will result in each pathway through it being exercised. Such data values can be inferred by identifying potential values in the scenario descriptions, and can then be inputted at run time into the system, whereby they will then be assigned to the respective input variables.

The scenarios being evaluated for the ATM system involve successful money withdrawal, and as such require suitable data arrangements to initialise and exercise these. The data chosen as is follows, card? = 101 , pin? = 5671 , and request? = 200.

After the selection of data for inputs, the specification can be executed. When execution is invoked, the ViZ system parses the ViZ/ZAL hybrid specification and presents a list of schemas that are identified. *Figure 4.8* shows the single schema ‘WithdrawMoney’ which can be selected to commence prototype execution. Subsequently, ViZ prompts the user to enter the input data. This is performed before the expression evaluation takes place. *Figure 4.9* shows an example of data entry, by demonstrating the input of the card number.

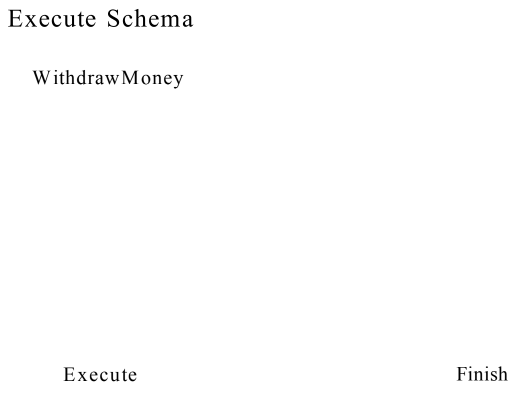
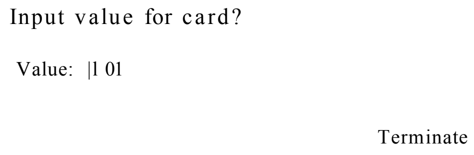


Figure 4.8. The ‘WithdrawMoney’schema as presented to the evaluators for execution.

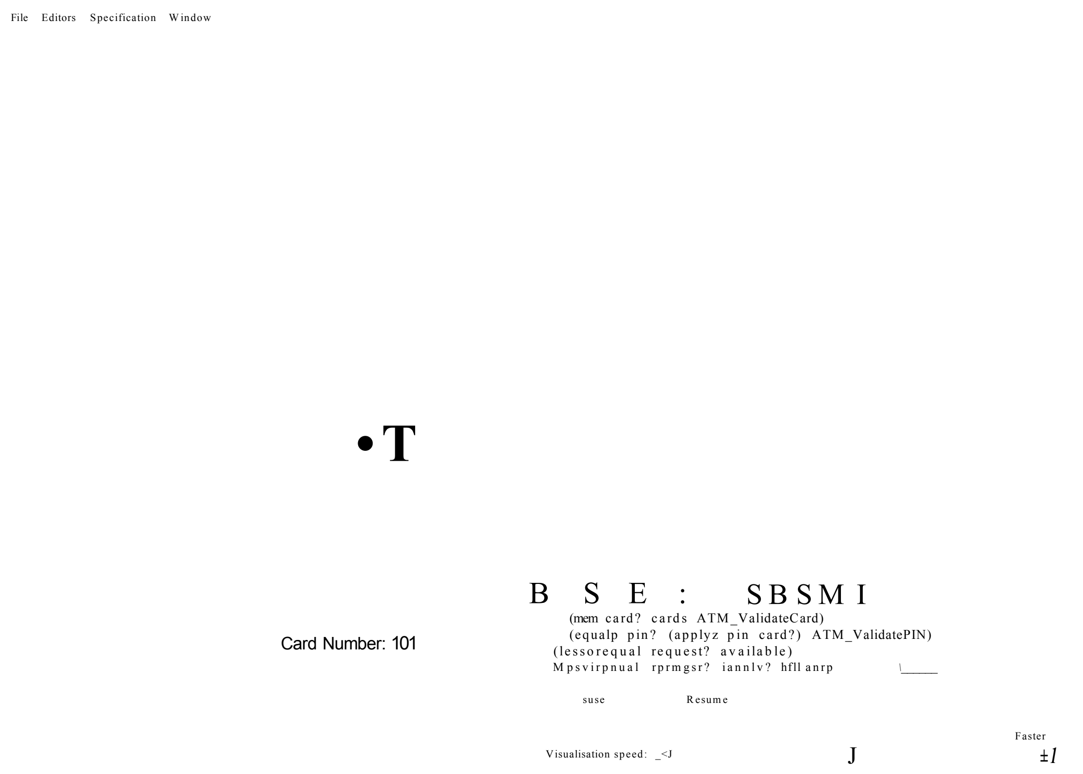

Input value for card?

Value: |1 01

Figure 4.9. Data entry prior to expression evaluation, showing the entry of the card details.

4.1.4 Results

Executing the ‘WithdrawMoney’ schema, using the given test data, results in a series of visualisations being generated. The visualisations generated for the execution of the first schema (that represents the insertion and validation of the user’s card) is shown in *Figure 4.9.* The visualisations represent Scenarios 1 and 2 in the ‘Successful Money Withdrawal’ respectively. For brevity however, the remainder of the visualisations are shown in *Appendix C.*



Card Number: 101

```
   (mem card? cards ATM_ValidateCard)
   (equalp pin? (applyz pin card?) ATM_ValidatePIN)
 (lessorequal request? available)
 Mpsvirpnual rprmgsr? iannlv? hfll anrp         \_____
```

```
   (mem card? cards ATM_ValidateCard)
   (equalp pin? (applyz pin card?) ATH_ValidatePIN)
 (lessorequal request? available)
 /1pqqnypmia 1 rpwipgf? t.annlv? ha Iftnrf r.ar.ri2J—1--------
```

Figure 4.9. Card insertion and validation.

4.1.5 Observations and Conclusion

The objectives of this particular demonstration of the ViZ system have been to show how it enhances the presentation of ZAL prototype execution, to show the invocation of the ViZ method and to illustrate the general capabilities of the system. To conclude this case study, each of these objectives will be discussed.

First, from the visualisations that result, it can be seen that the ViZ system harnesses and exploits visualisation to facilitate the graphical expression of ZAL prototype execution. The screenshots (shown in *Figure 4.9* and *Appendix* Q provide an example of the basic visual capabilities of the ViZ system. These capabilities are important in achieving the objective of overcoming the fundamental difficulties in presenting ZAL based prototype execution behaviour. The system enables visualisations to be constructed and attached, at strategic points, to a ZAL specification, and then interpreted and rendered by the software component of the ViZ system at run time. In doing this, this thesis argues that the resulting visualisations are more able to portray the execution behaviour in a customer comprehensible form than the original presentation generated by the ZAL system. In this instance, the visualisations, in the storyboard style, are able to portray the execution of the prototype in terms of the step-by-step usage of the ATM system. This style of presentation can be regarded as being more amenable to customer understanding.

It is also worth noting that enacting the method enabled certain features that were not in the original specification to be identified – for instance, discovering the need to accommodate card ejection. Conducting analysis at various points along the process provided greater opportunities for errors or omissions to be highlighted. This is an important and worthwhile characteristic of the ViZ process.

Second, this case study demonstrates the invocation and execution of the ViZ method. The method is designed to guide the developer through the process of visualisation development and application, by presenting the tasks to be undertaken in a structured manner. Importantly, the demonstration illustrated the enactment of the critical analysis phase of the process. This imposes a degree of rigour into the act of visualisation development by enabling information necessary for visualisation design to be elicited from specifications and corresponding scenario descriptions. It is this phase in particular that provides insight into the requirements of a proposed system, its specification, and how visualisations might be developed to portray its execution.

Third, the case study narrative and accompanying screenshots illustrate the general capabilities of the ViZ toolset. These include the tools to facilitate appearance and motion development, and scene construction, as well as an execution support system to facilitate visualisation rendering. The case study demonstrates how these tools are used together to develop complete visual prototypes.

To conclude, this case study has introduced and demonstrated the fundamental aspects of the ViZ approach. It has shown how visualisation is instrumental in transforming the mathematically based results of executing ZAL prototypes into a more comprehensible form that is amenable to customer comprehension in the context of customer validation, and importantly it has demonstrated the fundamental features of the system and how they are used.

4.2 Case Study – An Email System

This case study shows the application of the ViZ approach to a large industrial-scale project that involves prototyping the requirements for an email system. The objective is to illustrate how the method can be scaled, without modification, to accommodate complex systems.

4.2.1 Aims

The specific aims of this case study are as follows:

- To show how the application of the ViZ system remains simple despite the increase in size and complexity of the target specification.

- To demonstrate the design and development of non-trivial visual prototypes with a view to showing the ease and swiftness by which these can be constructed when the polymorphic and reusable characteristics of ViZ visualisations are exploited.

4.2.2 Context

The origins of the email system that is the focus of this case study can be found in [Cohen86] where a complete electronic office system is developed and described. The email system forms a major subset of this electronic office, which was originally specified using the Vienna Development Method (VDM-SL). It has since been translated into a specification based on the Z notation for use with the TranZit and ZAL tools. It forms an interesting case study for the ViZ system.

The email system provides facilities to enable a number of users to each possess an email account. In addition, the system provides features to enable users to compose and send emails, as well as receive, read, and reply to emails sent by other users.

To facilitate email operations, the user is supplied with an arrangement of 'trays' and a 'pad'. Trays are areas where emails are stored. There are three trays: an 'out-tray', where emails are stored that await sending after they have been composed; an 'in-tray', where incoming items from other users are stored; and a 'pending-tray' where copies of emails are stored that require replies. In the larger electronic office system, the 'pad' is a multipurpose editor/viewer. For the purpose of the email system, the pad is the area on which emails are composed or read. When a user has composed an email on the pad, they can transfer it to the out-tray in readiness for sending to the designated recipient(s). The out-tray can be flushed or 'cleared' to send the messages to the recipients, and at this point, the system should generate a time-stamp and a unique identifier for each email being sent. Emails that have been received from other users arrive in the in-tray, and can then be transferred to the pad for viewing. When a message is transferred to the pad, the current contents of the pad are overwritten. A sender can also demand that a reply to a message is needed. In this case, when the message is sent to the recipient's in-tray, an additional copy is made and placed in the recipient's pending tray that cannot be deleted until a reply is sent.

The system represents known users by the use of unique identifiers. However, while these are useful for the system they can be incomprehensible to potential users, therefore the system facilitates the use of aliases by which real human-readable names can be attributed to the internal identifiers used within the system.

The system is to provide the following functions to users:

- *Post.* Transfer the email on the user's pad to the out-tray. To be sent to the out-tray, the email must be well formed. A well-formed email comprises the attributes:
    - 'To list' – a list of recipients for that email.
    - "CC list" – a carbon-copy list (users who should also receive this message).
    - "Sender" – the name of the user sending this message.
    - "Subject" – A description of the contents of the message.
    - "Reply" – A flag indicating if a reply is required for this message.
    - "Reference" – Possible reference to one or more pending emails.
    - "Body" – The contents of the message.

  It is also a requirement to preserve the ordering of the names in the 'to list'.

- *Clear.* Transfer all documents in the user's out-tray to the recipient's in-trays, and possibly pending trays. Also generate time-stamps and unique identifiers.
- *List.* Display a summary of all items in the user's in-tray

- *Collect.* Transfer a particular email from the in-tray to the pad, overwriting the current contents of the pad and deleting the email from the in-tray.
- *Read.* Transfer an email from the pending-tray to the pad – do not delete until a reply is sent.
- *List Pending (ListP).* List a summary of emails in the pending tray.

In addition, three system administration activities are required:

- *Add User.* Create a new user account in the system.
- *Delete User.* Remove a user's account from the system.
- *Initialise (Init).* Initialise an empty email system.

The Z specification that corresponds to these requirements (which was derived from the original VDM-SL) can be seen in *Appendix D.*

4.2.3 Method

In accordance with the ViZ process, the three stages of scenario identification and documentation, visualisation design and construction and prototype execution and evaluation are enacted to develop the visual prototype. Specifically, the steps described below will be followed:

Stage 1 - Scenario Identification and Documentation

Step 1 Determine system functions of interest

Step 2 Identify and document scenario descriptions

Stage 2 - Visualisation Design and Construction

Step 3 Analyse scenarios to establish visualisation targets

Step 4 Develop a suitable visual metaphor/scheme for the whole visualisation

Step 5 Construct visual representations using the ViZ tools

Step 6 Apply visualisations to the ZAL specification to produce the visual prototype

Stage 3 - Prototype Execution and Evaluation

Step 7 Devise test cases from scenarios to guide the prototype evaluation

Step 8 Execute and evaluate the visual prototype

Scenario Identification and Documentation

Identifying and documenting scenarios commences with the decision as to which functions in the email system are to be validated or investigated further. For the purpose of this case study, the functions that will be investigated will be Post and Collect. By exploring the development of visualisations for more than one function, the ease of visual prototype development, as facilitated by reusable visual components, can be shown. The ZAL schemas that are of interest are shown below.

```
(SCHEMA POST
       :? ( uid? m? to? cc? from? subject? reply?.refs? body?)
       :PREDICATE
       (and
              (mem uid? (dom deskof))
              (not-mem m? mailitem)
              (not (equalp to? <>))
              (subset (ran to?) (dom (applyz direct uid?)))
              (subset (ran cc?) (dom (applyz direct uid?)))
              (make temp
                     { #((applyz deskof uid? )
                            (applyz outbox (applyz deskof uid? )))
                     })
              (setf mailitem' (unionz mailitem  {m?}))
              (setf mailto'
                     (unionz mailto
                            { #(m? (squash (resolve2 (mksi '#(k p) 'k
                            (dom to?) '(setf p (applyz deskof (applyz
                                   (applyz direct uid?) (applyz to? k))) )))))
                            }))
              (setf mailcc'
                     (unionz mailcc
                            { #(m? (squash (resolve2 (mksi '#(k p) 'k
                            (dom cc?) '(setf p (applyz deskof (applyz
                                   (applyz direct uid?) (applyz cc? k))) )))))
                            }))
              (setf mailfrom' (unionz mailfrom { #(m? from? ) }))
              (setf mailsubject' (unionz mailsubject { #(m? subject? )}))
              (setf mailreplyreq' (unionz mailreplyreq { #(m? reply? )}))
              (setf mailrefs' (unionz mailrefs  { #(m? refs? ) }))
              (setf mailbody' (unionz mailbody  { #(m? body? ) }))
              (setf outbox'
                     (override outbox
                            { #((applyz deskof uid?)
                            (unionz (applyz outbox (applyz deskof uid? ))
                            {m?})) }))
       )
)

(SCHEMA COLLECT
       :? ( uid? refno?)
       :INCLUDE delta_email
       :SHOW inbox
       :PREDICATE
       (and
              (mem uid? (dom deskof ))
              (mem refno? (ran mailrefno ))
              (setf m (applyz (inverse mailrefno ) refno? ))
              (mem m (applyz inbox (applyz deskof uid? )))
              (make temp
                     {#((applyz deskof uid? )
                            (applyz inbox (applyz deskof uid? )))
                     })
              (setf mail_subject (applyz mailsubject m))
```

```
             (setf mail_to (applyz mailto m))
             (setf mail_cc (applyz mailcc m))
             (setf mail_body (applyz mailbody m))
             (setf mail_from (applyz mailfrom m))
             (setf mail_replyreq (applyz mailreplyreq m))
             (setf mail_refs (applyz mailrefs m))
             (setf inbox' (override inbox
                                 { #((applyz deskof uid?)
                                        (setsub
                                        (applyz inbox (applyz deskof uid? ))
                                        {m})) }))
             (setf mailitem' (setsub mailitem {m}))
             (setf mailto' (domsub {m} mailto))
             (setf mailcc' (domsub {m} mailcc))
             (setf mailfrom' (domsub {m} mailfrom))
      )
)
```

Next, it is necessary to develop the corresponding scenario descriptions. Taking the Post operation first, it can be seen that this is concerned with taking the details of an email from the pad and storing these in the user's out-tray as a well-formed email. Such an operation requires the user's details as an input to the operation as well as the attributes that comprise an email. The operation first performs two pre-conditional checks; the first to determine the existence of the inputted user within the system (validate the user), and the second to see if the new email is not already part of the system as indicated by the existence of a unique identifier (validate the email). After this, the rest of the operation is concerned with adding the attributes of the email to the user's out-tray.

In operation there are two obvious outcomes, namely a successful Post execution, and an unsuccessful one. A successful Post operation is predicated on the user and the email being valid. If either of these is invalid then the Post operation should fail. Successful and unsuccessful Post operations can be characterised succinctly as:

$$\text{post success} \Rightarrow (\text{valid user} \wedge \text{valid email})$$

$$\text{post failure} \Rightarrow (\text{user not recognised} \vee \text{email not valid})$$

It can be seen that this basic analysis is performed in much the same way as in *Case Study 1 – the ATM System*. For systems with a discrete, and possibly small number of states, this approach is most successful.

The Collect operation is concerned with retrieving the details of a message from a given user's in-tray to the pad (and is therefore opposite to the Post function). This operation requires the user's ID to indicate which in-tray to collect the email from.

The first activity in Collect is to determine the existence of the user within the system (validate the user). After this, there is a check to determine if the given email is actually in the user's tray or not (validate the email). If these pre-conditions are satisfied, the email's details are transferred to the pad. Again, in practice, there are two possible outcomes, namely a successful and unsuccessful Collect invocation. These can be characterised as:

collect success ⇒ (valid user ∧ given email is in the user's tray)

collect failure ⇒ (user not recognised ∨ given email is not in the user's tray)

From this basic analysis, scenarios for both functions can be determined. At this time, just the Successful Post and Successful Collect scenarios will be focussed upon. The first step is to characterise the general behaviour of the functions. From the informal requirements, the behaviour of the Post and Collect operations are as follows:

Post:

1. Validate User ID
2. Validate Email ID
3. Customer enters email contents into the Pad
4. Email is sent, if the user ID and email ID are valid.

Collect:

1. Validate User ID
2. Validate Email ID
3. Retrieve specified email into the user's Pad.

The second step involves refining the generalised forms into ones that reflect the specific inputs that are required to produce the successful and/or unsuccessful results. For the Successful Post and Collect operations, these are:

Successful Post:

1. Customer enters User ID
2. User ID is validated successfully
3. Customer enters Email ID
4. Email ID is validated successfully
5. Customer enters email contents into the Pad
6. Email is sent

Successful Collect:

1. Customer enters User ID
2. User ID is validated successfully
3. Customer enters Email ID
4. Email ID is validated successfully Validate User ID
5. Retrieve specified email into the user's Pad.

The last step in developing scenario descriptions is to augment these activities with details elicited from the requirements with regards to AIOs, data, and system responses. These descriptions, expressed using the notation given by Regnell [Regnell95], can be seen in *Figure 4.16* and *4.17* respectively.

Once these scenario descriptions have been developed, visualisation design and construction may proceed.

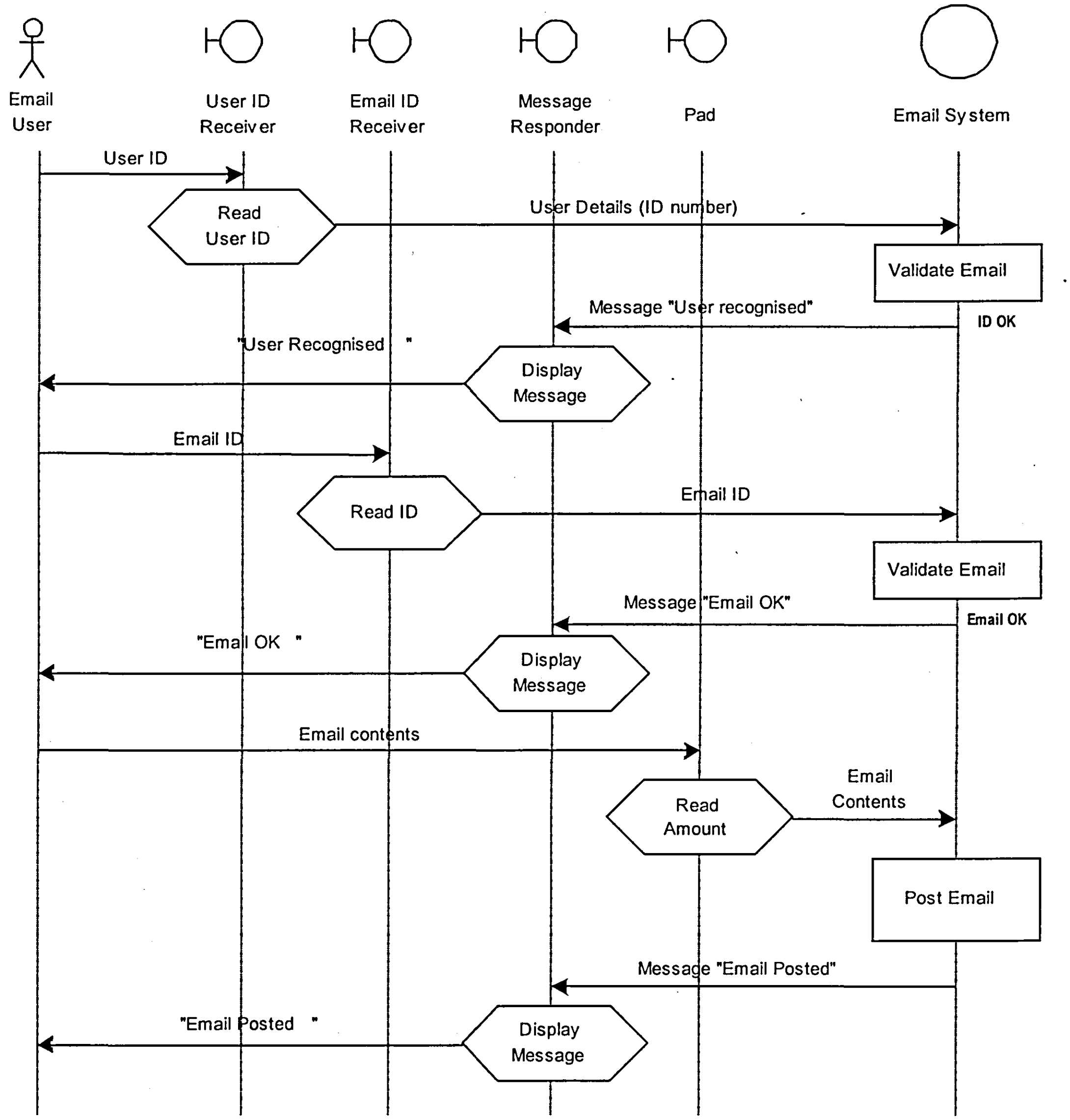


Figure 4.16. *Email System Scenario: Successful Post Operation.*

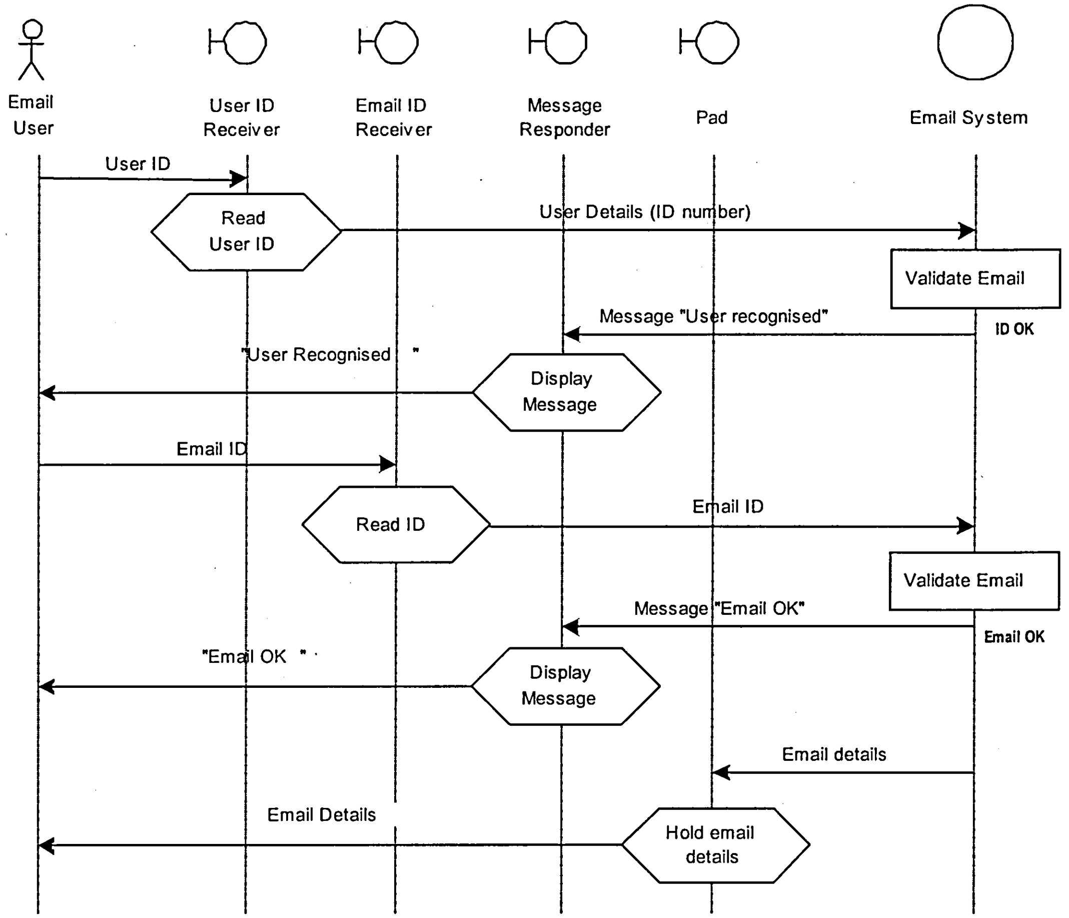


Figure 4.17. Email System Scenario: Successful Collect Operation.

Visualisation Design and Construction

Visualisation design and construction comprises four steps: scenario analysis, design, implementation, and association with the specification (as described *Section 3.2.2*).

The first task is to analyse the scenarios to first establish a structure for the visualisation, expressed in terms of scenes that will portray the execution of expressions then establish the contents of the scenes by identifying potential visualisation targets.

The scenes that may be used to visualise the Successful Post and the Successful Collect operations are shown in *Tables 4.5* and *4.6* respectively.

Scenario activity	Possible scenes	Related ZAL expression
1. User enters their ID number 2. User is validated	Scene 1. Show a user entering their ID number (used to show the state of the system before executing the expression). Scene 2. Show results of user ID validation (used to show the state of the system after the expression is executed).	(mem uid? (dom deskof))
3. User enters the new email's ID 4. Email is validated	Scene 3. (before state) Show a user entering the ID of a new email Scene 4. Show results of email validation	(not-mem m? mailitem)
5. User enters email's attributes and contents into the pad	Scene 5. Show the contents of the Pad after all email details have been entered. (Visualise the email attributes).	(setf mailitem' (unionz mailitem {m? })) ...... (etc)
6. Post email (i.e. transfer it to the out-tray) and inform the user of success.	Scene 6. Show the contents of the out tray with the new email in it.	(setf outbox' (override outbox { #((applyz deskof uid?)(unionz (applyz outbox (applyz deskof uid?)) {m? })) }))

Table 4.5. Correspondence between scenarios and expressions, and possible scenes for the Successful Post scenario.

Scenario activity	Possible scenes	Related ZAL expression
1. User enters their ID number 2. User is validated	Scene 1. Show a user entering their ID number Scene 2. Show results of user validation	(mem uid? (dom deskof))
3. User enters the new email's ID 4. Email is validated	Scene 3. Show a user entering the ID of a new email Scene 4. Show results of email validation	(mem m (applyz inbox (applyz deskof uid?)))
5. Transfer contents of email to the user's Pad	Scene 5. Show the contents of the Pad after all email details have been transferred.	(setf mail_subject (applyz mailsubject m)) ..... (etc)

Table 4.6. Correspondence between scenarios and expressions, and possible scenes for the Successful Collect scenario.

Furnishing this structure with details about the potential contents is achieved by examining the scenarios and executable specification to identify features of interest that should be visualised. Such features include actors, abstract interface objects, and data elements, and are gleaned by examining each activity in the scenarios in turn. The complete set of features that should be visualised for the Successful Post scenario and the Successful Collect scenario are shown in *Tables 4.7* and *4.8*, respectively.

Successful post operation		
Actors	AIOs	Data
1. User enters their ID number		
Email User	User ID receiver	User ID number
2. Validate user		
Email User	Message responder	"User OK" (message)
3. User enters the new email's ID		
Email User	Email ID receiver	Email ID number
4. Email is validated		
Email User	Message responder	"Email OK" (message)
5. User enters email's attributes and contents into the pad		
Email User	Pad	Email attributes
6. Post email and Inform user of successful post		
Email User	Message responder	Contents of the user's out-tray

Table 4.7. *Activities and important elements for the scenario for Successful Post.*

Successful Collect Operation		
Actors	AIOs	Data
1. User enters their ID number		
Email User	User ID receiver	User ID number
2. Validate user		
Email User	Message responder	"User OK" (message)
3. User enters the new email's ID		
Email User	Email ID receiver	Email ID number
4. Validate email		
Email User	Message responder	"Email OK" (message)
5. Transfer email details to the Pad		
Email User	Pad	Email attributes

Table 4.8. *Activities and important elements for the scenario for Successful Collect.*

As in the analysis conducted in *Case Study 1*, this process provides much insight. Not only does it reveal a scene-based structure of a visualisation, but it also indicates where visualisation effort should be concentrated to produce user-centric portrayals of the behaviour of the prototype.

Visual Metaphor Design

The next design task is to devise a visual metaphor upon which the actual format and style of the visualisations can be based. This is achieved by using the information about the structure and composition previously derived. Once again, the key objective is to develop a visual style that enables the meaning of the underlying requirements to be portrayed in a fashion that the user can recognise.

In terms of the Email System, a suitable metaphor could consist of a visualisation of the Pad, as this is focus of the user's interaction with the system. A visualisation of the contents of the relevant trays (the 'out-tray' and the 'in-tray' for the Post and Collect operations respectively) would also be useful. These visualisations correspond to the elements revealed from the previous analysis phase.

By applying the resulting 'visual concept' a simple design scheme can be developed. Using the 'User ID Number' that features in both the Successful Post and Successful Collect scenarios as a design example (from *Tables 4.8* and *4.9*), it can be seen this is a data item that represents a numeric value within the executable formal specification. A visual representation of the user ID might be best portrayed using a direct display of the value. By using a direct representation of the numeric value, it can be argued that its meaning within the prototype can be made overt and eliminate the need, on the part of the viewer, to perform some mental translation from an abstract form. Similar treatments can be applied to the other elements that should feature in the visualisation.

Table 4.9 shows conceptual visualisations for the entities discussed above, populated with example data values to show how the possible relationships between the graphical and textual items.

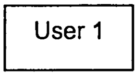
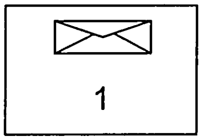
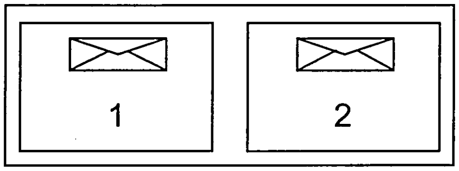
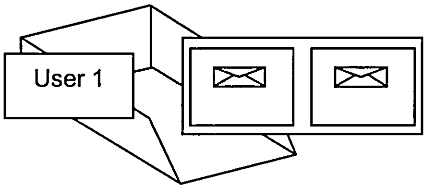
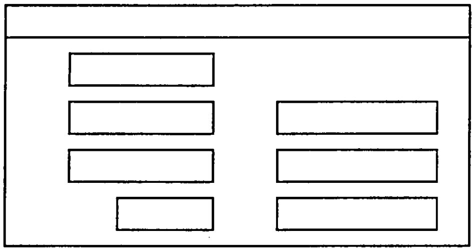
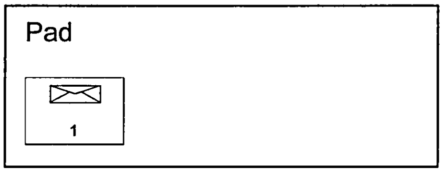
Entity	Data Type	Anticipated Representation
User ID	Displays an integer appearance	
Single Email Representation (text value augmented with image of email)	Displays a text value	
Tray Contents (Collection of emails, uses a series of individual email representations)	Displays a set	
Complete Tray (represents both the tray's owner – user ID and the corresponding tray contents)	Displays the relationship between user and tray.	
Complete Email (represents the attributes of a complete email message – complete with boxes in which the details will be placed).	Displays a combination of integer and text values.	
The Pad (depicts the user's own desktop pad)	Displays a pad with/without email.	

Table 4.9. *Possible representations for the entities involved in email system.*

Visualisation Construction

Construction of the visualisations using the ViZ tool follows design. This involves construction of visualisation objects and then construction of complete scenes containing the visualisation objects. The development of a complete scene was described in detail in Case Study 1, and therefore a similar treatment is not necessary here. Instead, the reuse of visualisation objects to demonstrate their polymorphic nature will be elaborated upon.

First, suitable containers must be created in the visual component library. These are used to store the respective visualisations as they are created with the ViZ tools. Next, appearances that portray the pertinent elements of the scenarios (email, trays, and pad) are constructed using the ViZ Appearance Editor. The final forms of the appearances, with sample data are shown in *Table 4.10.*

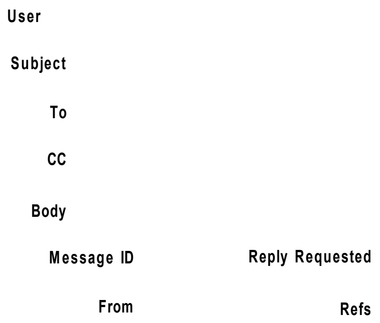
Purpose	Representation	Identifier
To represent the system state variables: InBox, OutBox, Pending.		**Tray Appearance**
Used to encapsulate the attributes of an email.	 User Subject To CC Body Message ID Reply Requested From Refs	**Email Appearance**
Represents the user's pad, complete with email identifier.		**Pad Appearance**

Table 4.10. Final form of the appearances for the elements in the email case study.

Spatial and positional information for each visualisation object must be specified through the use of the ViZ Motion Editor. For this particular case study, it is not necessary to specify animation/movement, but merely to indicate the on-screen locations of the visualisation objects. Again, these components are stored in the component repository.

Complete scenes must be defined, using the VIZ Visualisation Editor. Scenes specify the visualisation objects that should be used to visualise particular elements of an expression. The scenes developed should correspond with those indicated earlier in the process.

However, once visualisation objects, which are common to one or more scenes, have been developed, their polymorphic characteristics can be exploited to deliver large improvements in productivity to reduce development time. Polymorphism is manifested in visualisation objects through the typing system, that is, visualisation objects are designed to be 'type-oriented' but not 'semantically-oriented'. In other words, the application of visualisation objects to execution artefacts, such as the data in an executable specification, does not rely on any semantic information from the specification (or the underlying requirements) other than the type of the entity being visualised. This enables a visualisation object that is designed for visualising a set, for instance, to be applied to any set in any specification. Whether the application is appropriate is a separate matter. Appearances for other types act similarly. The effect of this, in terms of the Email System example, is that a visualisation object that is designed to represent a generic tray can be used for all tray variants – i.e. in-tray-, out-tray and pending tray. This facility, although simple, can be used to great effect when developing separate scenes that contain common entities. In addition to polymorphism, the visual component library is instrumental in facilitating reuse through the storage of all appearance, dynamic, and scene components for subsequent re-application in other validation projects.

From *Tables 4.7* and *4.8*, the dependencies and reuse opportunities that are exploited in the Success Post and Successful Collect scenarios can be seen. It can also be seen that the actor 'Email User' features in all the scenes used in the portrayal of the scenarios, and consequentially, the same visualisation object is used. Similar treatments for Abstract Interface Objects and data elements are employed.

The ViZ visualisation editor is applied, at this point, to combine the visualisation objects into scenes. The scenes created for the Successful Post and Successful Collect scenarios are shown in *Table 4.11.*

Scene Name	Purpose
Email_Vis	Visualise a well formed email, complete with input parameters.
InTray_Vis	Visualise the emails in the In Tray
OutTray_Vis	Visualise the emails in the Out Tray

***Table 4.11**. Scenes and identifiers for the Post and Collect prototypes.*

Apply Visualisations to Specification

The fourth and final step in visualisation design and construction is to append references to complete scenes to the appropriate specification sections. This also requires suitable type information to be superimposed over the underlying ZAL executable specification. The result is the hybrid ZAL-ViZ schemas shown below.

```
(schema Post
    (header
        local mailitem'
        local mailto'
        local mailcc'
        local mailfrom'
        local mailsubject'
        local mailreplyreq'
        local mailrefs'
        local mailbody'
        local outbox'
        local k
        local p
    )
    (and
        (show uid? subject? to? cc? body? m? from? reply? refs?
            Email_Vis)
        (mem uid? (dom deskof))
        (not-mem m? mailitem)
        (not (equalp to? <>))
        (subset (ran to?) (dom (applyz direct uid?)))
        (subset (ran cc?) (dom (applyz direct uid?)))
        (make temp
            { #((applyz deskof uid? )
            (applyz outbox (applyz deskof uid? ))) })
        (show temp m? OutTray_Vis)
        (setf mailitem' (unionz mailitem  {m?  }))
        (setf mailto' (unionz mailto
            { #(m? (squish (mksi '#(k p) 'k (dom to?) '(setf p (applyz
            deskof (applyz (applyz direct uid?) (applyz to? k))) ))))
            }))
```

```
             (setf mailcc' (unionz mailcc
                 { #(m? (squish (mksi '#(k p) 'k
                 (dom cc?) '(setf p (applyz deskof (applyz
                 (applyz direct uid?) (applyz cc? k))) ))) ) }))
             (setf mailfrom' (unionz mailfrom { #(m? from? ) }))
             (setf mailsubject' (unionz mailsubject  { #(m? subject? ) }))
             (setf mailreplyreq' (unionz mailreplyreq  { #(m? reply? ) }))
             (setf mailrefs' (unionz mailrefs  { #(m? refs? ) }))
             (setf mailbody' (unionz mailbody  { #(m? body? ) }))
             (make temp { #((applyz deskof uid? )
                 (unionz (applyz outbox (applyz deskof uid? )) {m?})) })
             (show temp OutTray_Vis)
             (setf outbox' (override outbox  { #((applyz deskof uid? )
                 (unionz (applyz outbox (applyz deskof uid? )) {m?})) }))
         )
     )

     (schema Collect
         (header
             local mailitem'
             local mailto'
             local mailcc'
             local mailfrom'
             local mailsubject'
             local mailreplyreq'
             local mailrefs'
             local mailbody'
             local inbox'
         )
         (and
             (mem uid? (dom deskof ))
             (mem refno? (ran mailrefno ))
             (setf m (applyz (inverse mailrefno) refno? ))
             (mem m (applyz inbox (applyz deskof uid? )))
             (make temp { #( (applyz deskof uid? )
                 (applyz inbox (applyz deskof uid? ))   ) })
             (show temp m InTray_Vis)
             (setf mail_subject (applyz mailsubject m))
             (setf mail_to (applyz mailto m))
             (setf mail_cc (applyz mailcc m))
             (setf mail_body (applyz mailbody m))
             (setf mail_from (applyz mailfrom m))
             (setf mail_replyreq (applyz mailreplyreq m))
             (setf mail_refs (applyz mailrefs m))
             (show uid? mail_subject mail_to mail_cc mail_body m mail_from
                 mail_replyreq Email_Vis)
             (setf inbox' (override inbox  { #((applyz deskof uid? )
                 (setsub (applyz inbox (applyz deskof uid? )) {m})) }))
             (setf mailitem' (setsub mailitem {m}))
             (setf mailto' (domsub {m} mailto))
             (setf mailcc' (domsub {m} mailcc))
             (setf mailfrom' (domsub {m} mailfrom))
         )
     )
)
```

Prototype Execution and Evaluation

The final stage in the ViZ process facilitates the evaluation of the visual prototype. For execution to take place, test data must be prepared that will enable the scenarios to be exercised.

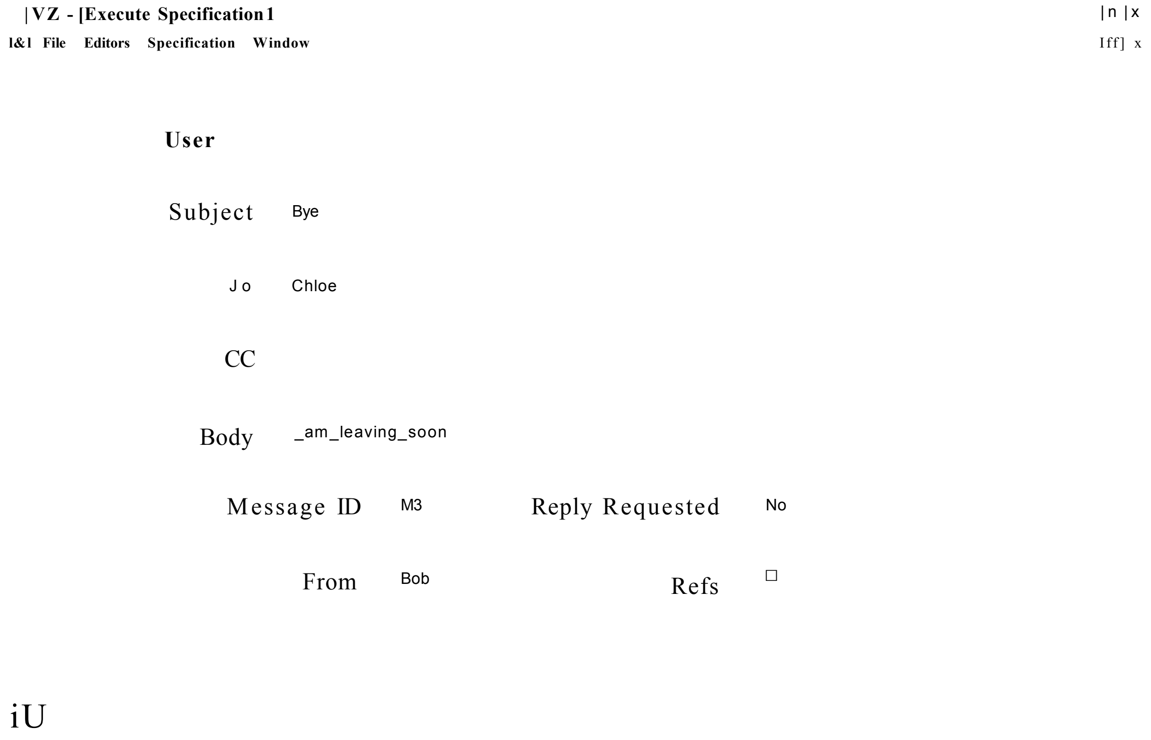
The test data required to exercise both the Successful Post and Collect scenarios can be derived directly from the system state definitions in the ZAL executable specification, and is shown in *Table 4.12*.

	Input	Value
Successful Post	uid?	1
	subject?	'Meeting
	to?	<'Chloe>
	cc?	< >
	body?	'What_time?
	m?	'_
	from?	'Bob
	reply?	'No
	refs?	{ }
Successful Collect	uid?	1
	refno?	102

Table 4.12. *Test data required for Successful Post and Collect schema executions.*

4.2.4 Results

The results of executing the Email System prototype are presented below in *Figures 4.18* and *4.19*. *Figure 4.18* shows the results of executing the Post specification, the aim of which was to demonstrate the visualisation of a well-formed email. *Figure 4.19* shows the visualisations produced by executing the Collect specification. This retrieves the given email (reference number 102) from the user's in-tray and places it on the user's pad. The screenshots show first the existence of email number 102 in the user's in-tray, and second, the details of email 102 on the user's pad.


|VZ - [Execute Specification1
I&l File Editors Specification Window
|n |x
Iff] x
iU

User

Subject Bye

J o Chloe

CC

Body _am_leaving_soon

Message ID M3 Reply Requested No

From Bob Refs □

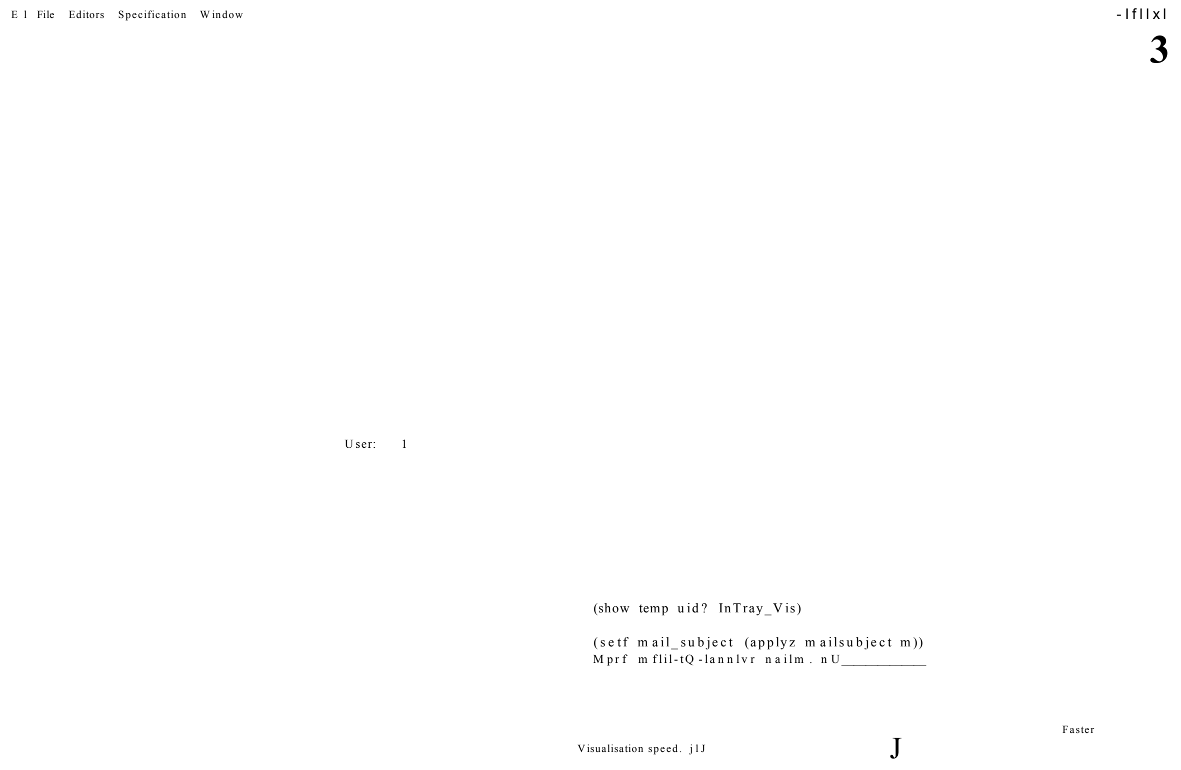
Figure 4.18. Visualisation results for the Successful Post scenario.

4.2.5 Observations and Conclusions

This case study has highlighted the capabilities of the ViZ system with regards to scalability. It showed how the ViZ method is applied, without change, to larger systems. The only aspect where change could occur would be that certain activities are repeated to take into account the greater number of visualisations that are required.

This case study has also illustrated the utility of the reusable qualities of the visual representations that can be created using the ViZ system. This, in turn, is coupled with the scalability issue, in that reuse enables visual prototypes for large system to be developed without greatly increasing the effort or time overhead required.

Although useful, reuse of visual components is not without its drawbacks. The main difficulty is that it provides more opportunities for applying inappropriate visualisations. The type system ensures that visualisations for different types of system state variables are not applied, and so goes some way to alleviate this problem, but it does not prevent visualisations for the same type being applied.

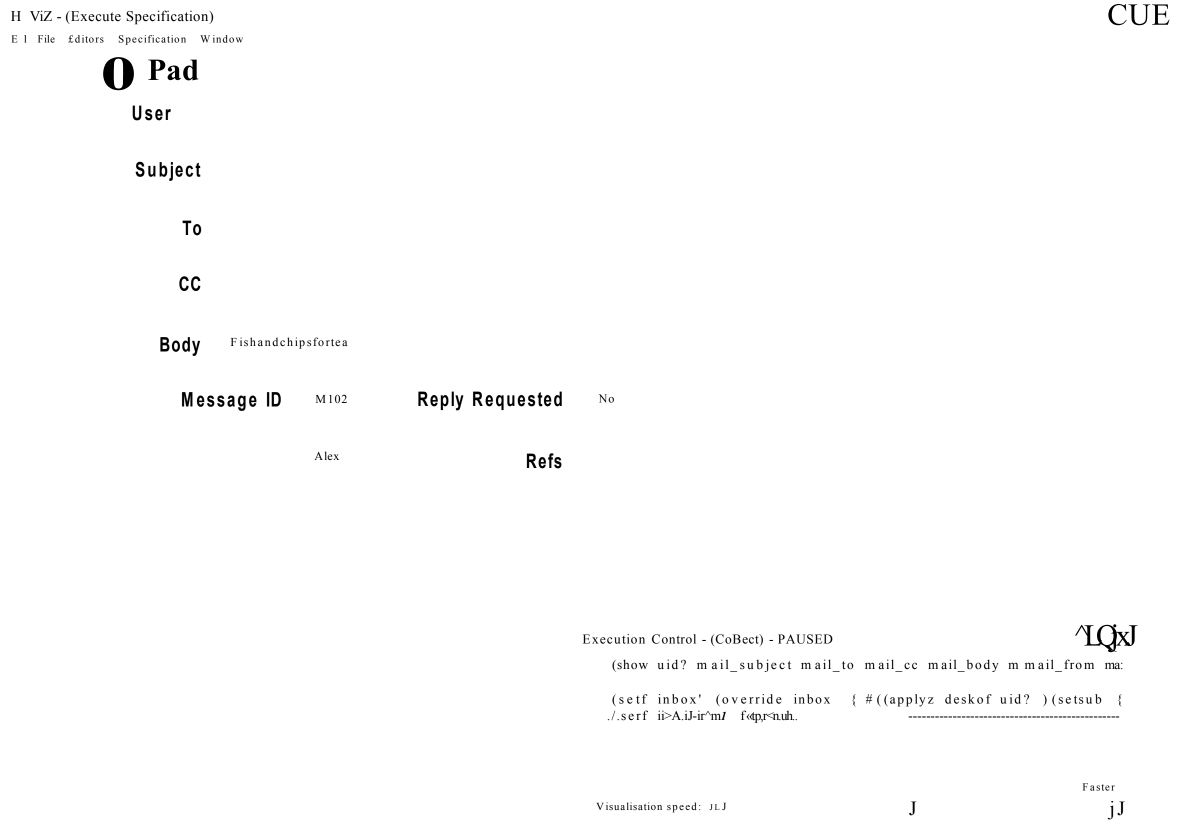


User: l

```
(show temp uid? InTray_Vis)

(setf mail_subject (applyz mailsubject m))
Mprf mflil-tQ-lannlvr nailm. nU___________
```

Visualisation speed. jlJ


H ViZ - (Execute Specification)
CUE
E I File £ditors Specification Window
O Pad
^LQjxJ
Faster
J
jJ

User

Subject

To

CC

Body Fishandchipsfortea

Message ID M102 Reply Requested No

Alex Refs

Execution Control - (CoBect) - PAUSED

```
(show uid? mail_subject mail_to mail_cc mail_body m mail_from ma:

(setf inbox' (override inbox { #((applyz deskof uid? )(setsub {
./.serf ii>A.iJ-ir^m1 f«tp,r<n.uh..              ------------------------------------------------
```

Visualisation speed: JLJ

Figure 4.19. Visualisation results for the Successful Collect scenario.

4.3 Case Study – A Water Level Monitoring System

This case study (documented in [Ozcan98b]) demonstrates the development of a prototype for a system in the domain of safety critical applications.

4.3.1 Aims

The central aim of this case study is to demonstrate the correspondence preserving mechanism that are offered by the ViZ system. These mechanisms offer assurance that visual representations used to portray the execution of the specification are used appropriately and are used in the manner for which they were created. Developers or visualisation designers create visual representations with the intent that they will have a specific purpose – often they will portray visually particular execution artefacts or specific elements in a visualisation and often in a specific context. The developers thus give the representation an inherent meaning, and although this meaning might not specified using identifiable or formal attributes, it still forms part of the visualisation nonetheless. If the representation is applied or subsequently reused to portray a different execution artefact, and if this execution artefact does have a different meaning, then the visualisation may not correspond to what is intended, resulting in the potential for misinterpreting the visual prototype.

This case study shows how two simple mechanisms can be used to prevent misapplication of visual representations so that correspondence is preserved, i.e. visual representations are not applied, either intentionally or accidentally, to execution artefacts with meanings/purposes different to those that were initially intended.

The case study will further illustrate the utility of the ViZ system in general, by presenting another example of how visualisations of the behaviour of a system can be developed and used.

To demonstrate these points, the case study focuses on the development of a visual prototype for a substantial industrial-scale safety critical application. The system is a water level monitoring system, the requirements of which are described in the following section.

4.3.2 Context

The water level monitoring system (WLMS) is a safety critical application used in the domain of steam generation, for example, in power generation or heating systems [Jackson93, VanSchouwen91, Williams94]. The Z specification for the WLMS is presented in *Appendix E.* The WLMS is intended to provide human operators with information as to the state of the steam generation system and also to provide a simple control system that is employed in the event of an error or system failure.

Physically, the WLMS is applied to a hardware system consisting of two water vessels (see *Figure 4.20).* During steam generation, water is pumped from the reservoir to the steam generator where it is boiled. The steam's destination is not relevant to this case study.

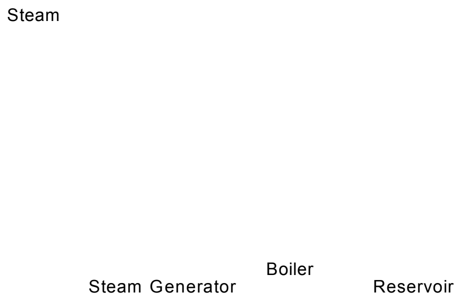
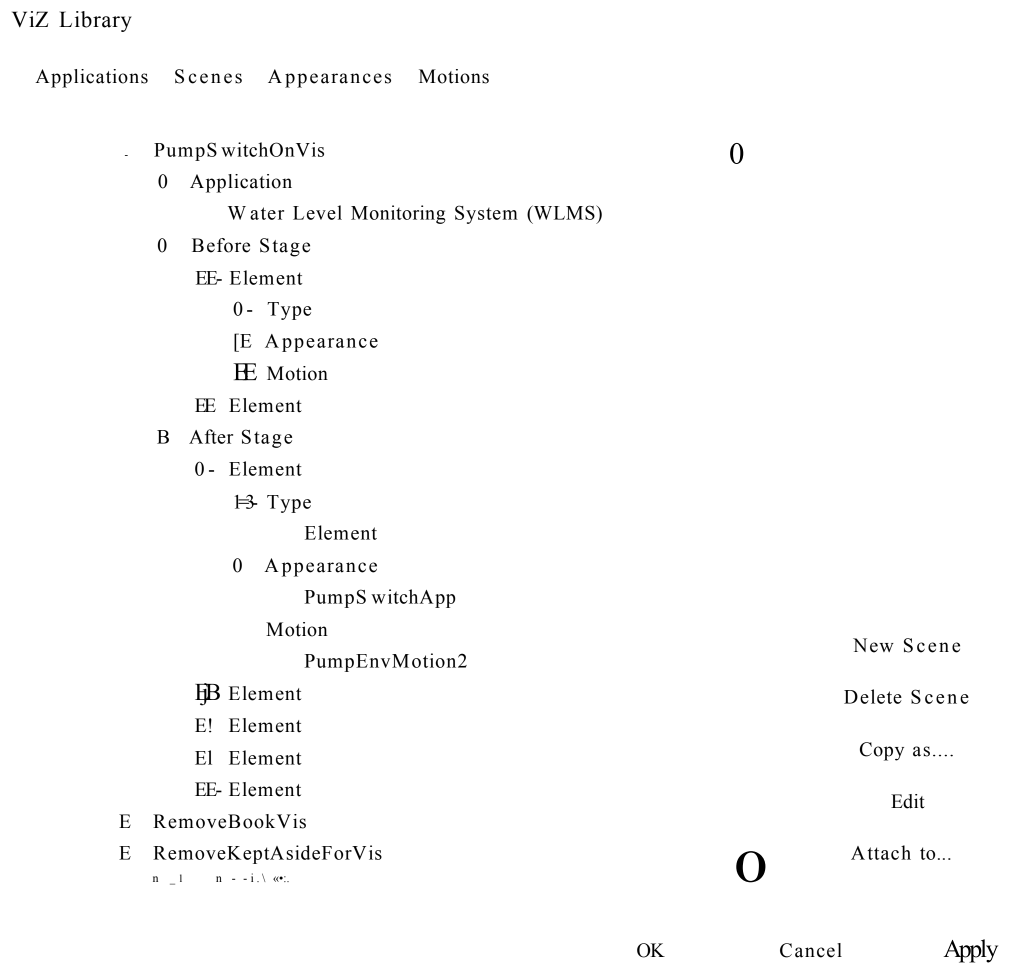


Figure 4.20. A typical steam generation system.

The WLMS monitors the water level in the reservoir and sounds an alarm if it becomes too low or too high. The pump is shut down should this situation occur. The pump is also turned off if the WLMS control unit itself fails. In both cases, switching the power supply off turns off the pump.

The WLMS has two push buttons. The first is a 'self test' button that lets the operator check the system while it is shut down, the second is a 'reset' button that is used to bring the system back to normal operation following a shut-down or test as long as the water level is within the safe range. Internal faults in the WLMS are detected by an external 'watchdog' that receives a periodic signal from the WLMS. If the signal is not received, the watchdog assumes the WLMS has failed and turns off the water pump. In addition, a global clock provide a time signal to co-ordinate system shown in a non-graphical view in the ViZ visualisation repository, in *Figure 4.27.* This will be attached, along with type information, to the PumpEnvironment schema in the executable specification, to form a hybrid ZAL/ViZ specification. This PumpEnvironment schema is shown below.



```
. PumpSwitchOnVis
  0  Application
        Water Level Monitoring System (WLMS)
  0  Before Stage
     EE- Element
        0- Type
        [E Appearance
        EE Motion
     EE Element
  B  After Stage
     0- Element
        E3- Type
              Element
        0  Appearance
              PumpSwitchApp
           Motion
              PumpEnvMotion2
     EjB Element
     E! Element
     El Element
     EE- Element
E  RemoveBookVis
E  RemoveKeptAsideForVis
   n _l    n - -i.\ «•:.
```

Figure 4.27. The completed scenefor the Pump Environment visualisation.

```
(schema PumpEnvironment
       (and
              (if     (and
                             (eq powerNow?  'on)
                             (eq shutdownSignal  'go)
                             (eq watchdog!  'operate)

                     then    (make pumpSwitch!  'closed
                              PumpSwitchOnVis  shutdownSignal  watchdog!
                             powerNow? waterLevel)
                     else
                             (truev)


              (if     (or
                             (eq powerNow?  'off)
                             (eq shutdownSignal  'stop)
                             (eq watchdog!  'shut)
```

```
                then    (make pumpSwitch!  'open PumpSwitchOffVis
                         shutdownSignal watchdog! powerNow? waterLevel)
                else    (truev)
```

At this time, the specification can be executed to produce the simulation of the Pump Environment schema. As with the previous case studies, appropriate test data can be derived from the scenario descriptions to drive the evaluation process (although this will not be covered in this case study).

4.3.4 Results

The result of executing the simulation of the Pump Environment function is shown in *Figure 4.28.* The execution is performed with the conditions that result in the pump operating normally, i.e. the power is on, the shut down signal is 'go' and watchdog is 'OK', with the visualisation reflecting the status of the system.

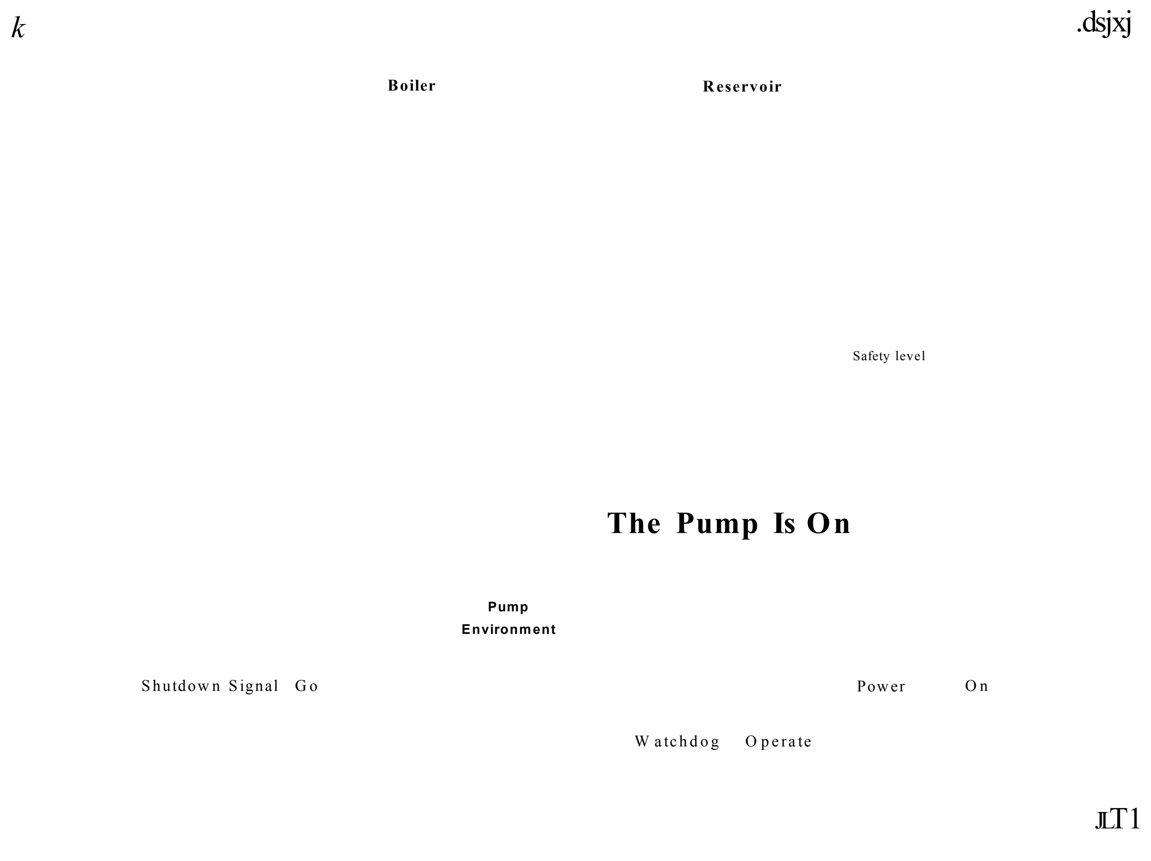


Figure 4.28. The results of executing the Pump Environment simulation.

4.3.5 Observations and Conclusions

To conclude this case study, a demonstration of the effectiveness and utility of the correspondence preserving mechanisms will be described. These mechanisms consist of the type system, which prevents visualisations designated for one type of system state variable being applied to variables of other types, and the constraints that form part of the visualisation editors that prevent visualisations designed for other validation projects being applied to the current one. These mechanisms will be demonstrated through the development of a visualisation for an additional WLMS function. In concert, a discussion of the activities required (and their implications) when re-applying visual representations and how the correspondence preserving mechanisms facilitate this, will also be given.

The function chosen as the vehicle to demonstrate the correspondence preserving mechanisms is 'PumpControl'. This determines the status of the ShutdownSignal, which in turn is used by the PumpEnvironment function to set the state of the pump switch. The ShutdownSignal is set according to the conditions in *Table 4.14* (these conditions are abstracted from the description of the system given in *Section 4.3.2*):

Operating Mode	Failure Mode	Reset Button	Shutdown Signal
Operating	AllOk	Not Pressed	Go
Operating	AllOk	Pressed	<unchanged>
Shutdown	AllOk		<unchanged>
Standby	AllOk		Stop
Test	AllOk		Stop
	Bad Level Device		Stop
	Hard Fail		Stop

Table 4.14. *Conditions used to determine the status of he ShutdownSignal*

By conducting a very rudimentary analysis, it can be seen that seven specific scenarios can be derived, each one corresponding to a row in *Table 4.14*, and any of which could be used as the target for visualisation. However, when developing these visualisations, it may be beneficial to consider the general case, and the elements that may require visualising in order to portray all permutations and combinations of the derived scenarios. Developing generalised visualisations 'up-front' is useful as the resulting scenes can accommodate and visualise a variety of scenarios without requiring change or subsequent effort later in the validation process. A generalised visualisation of the PumpControl function would at least require visualisation objects to depict each of the system state variables involved – in this case, the three input variables OperatingMode, FailureMode, and ResetButton, and the output variable ShutdownSignal.

After deciding which elements of executing the PumpControl schema should be visualised, the structure of the visualisation should be considered. Like many of the functions in the WLMS system, PumpControl is not actually invoked by an operator, but instead forms part of a wider arrangement of functions that contribute to the overall operation of the system. Inputs are collected without user intervention and passed as parameters to the function, and the single output is passed on to the PumpEnvironment function. Therefore, it is necessary to simply show the relationship between inputs and outputs, and this can be performed using a single visualisation scene that is shown after the schema has executed to reflect the final contents of the relevant system state variables.

This leads to the implementation of the scene using the ViZ tools, and for this particular scene, the reuse of visualisation objects. The scene must encapsulate visualisation representations for the system state variables identified above. These variables are all of type text string (known in ViZ as an 'element'). A text string visualisation already exists in the WLMS validation project and can be reused readily in this new context - the polymorphic nature of such a visualisation enables this. On completion of this construction activity, the scene shown in *Figure 4.29* is produced.

The correspondence preserving mechanisms come into play during the construction of scenes such as this. During construction, the developer/visualisation designer is required to specify the system state variables that the scene should visualise, along with the type of those variables (these details are shown in the four boxes titled 'Variable' in *Figure 4.29*). Specifying variables and their types is performed using the dialog box shown in *Figure 4.30*.

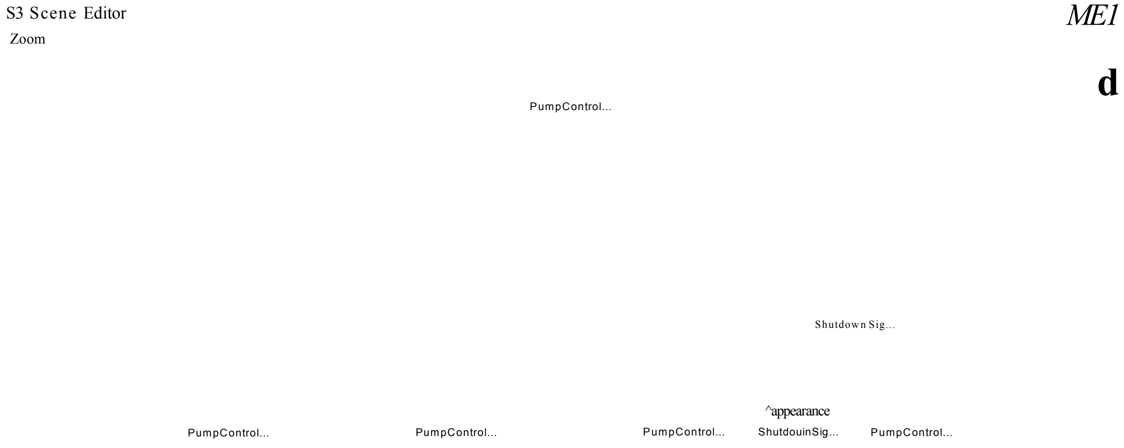
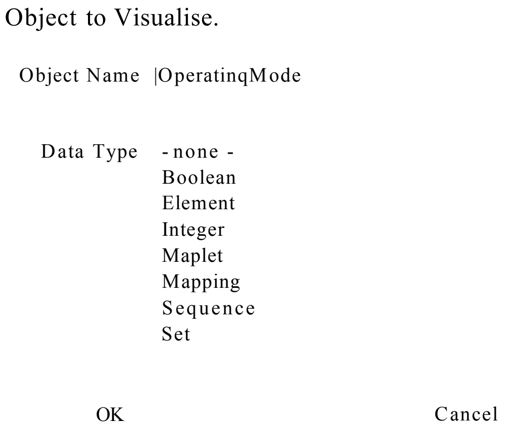


Figure 4.29. The contents of the scene to visualise to the PumpControl function.

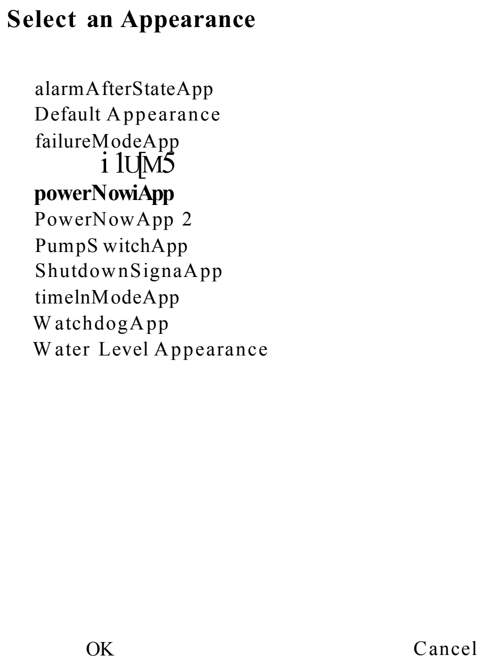
Specifying variables and their types is the first step in constructing scenes. The second step involves assigning appearances to them. When the developer/visualisation designer attempts to assign an appearance, they are presented with a list of representations that belong to the current validation project and are invited to select one *{Figure 4.31).* However when appearances are developed, using the ViZ Appearance Editor and prior to the scene construction stage, the designer attributes them with a data type. The data type of the appearance is thus checked against the one designated to the target variable in the scene. If they are equal, the selected appearance is associated with the variable. If not, the error message shown in *Figure 4.32* is displayed.


Object to Visualise.

Object Name |OperatinqMode

Data Type
- none -
Boolean
Element
Integer
Maplet
Mapping
Sequence
Set

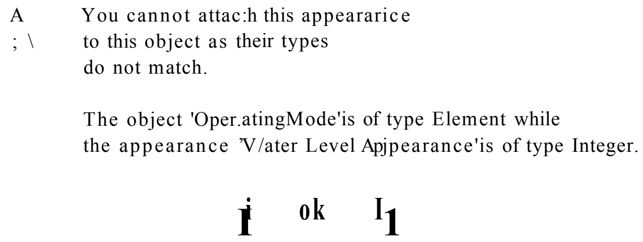
Figure 4.30. The dialog box used to specify variables and types that should be visualised by the scene.



Select an Appearance

alarmAfterStateApp
Default Appearance
failureModeApp
i lU[M5
powerNowiApp
PowerNowApp 2
PumpS witchApp
ShutdownSignaApp
timelnModeApp
WatchdogApp
Water Level Appearance

Figure 4.31. Appearance selection.



A You cannot attac:h this appeararice
; \ to this object as their types
do not match.

The object 'Oper.atingMode'is of type Element while
the appearance 'V/ater Level Apjpearance'is of type Integer.

Figure 4.32. Error message displayed when a type mismatch has been detected.

This simple checking mechanism limits the application of representations that are not of the same type, and thus presents a first line defence against accidental application. This is effective, as much thought is often imparted both before and during visualisation development as to the types of the representations and their use. The type system ensures that this forethought is not merely discarded.

The second correspondence preserving mechanism is that which ensures scenes, appearances and motions belonging to the same validation project can only be used together. This prevents the mismatching or inappropriate application of visualisation artefacts. This mechanism is facilitated through the use of ‘Application Containers’ in the ViZ repository, which are used to store all the visualisation artefacts created during the process of developing a visual prototype for a particular system. *Figure 4.33* shows the contents of the WLMS application container. Inside each application container, partitions exist for separation and storage of appearance and motion components, as well as complete scene-level visualisations. When selecting appearances or motion components to apply in the scene, the Scene Editor presents only those that belong in the container of the current development project. Appearances and motion components are partitioned at their time of implementation as to which container they should be associated. This mechanism provides a simple, yet effective approach for preventing visual representations being applied in situations where they might be inappropriate.

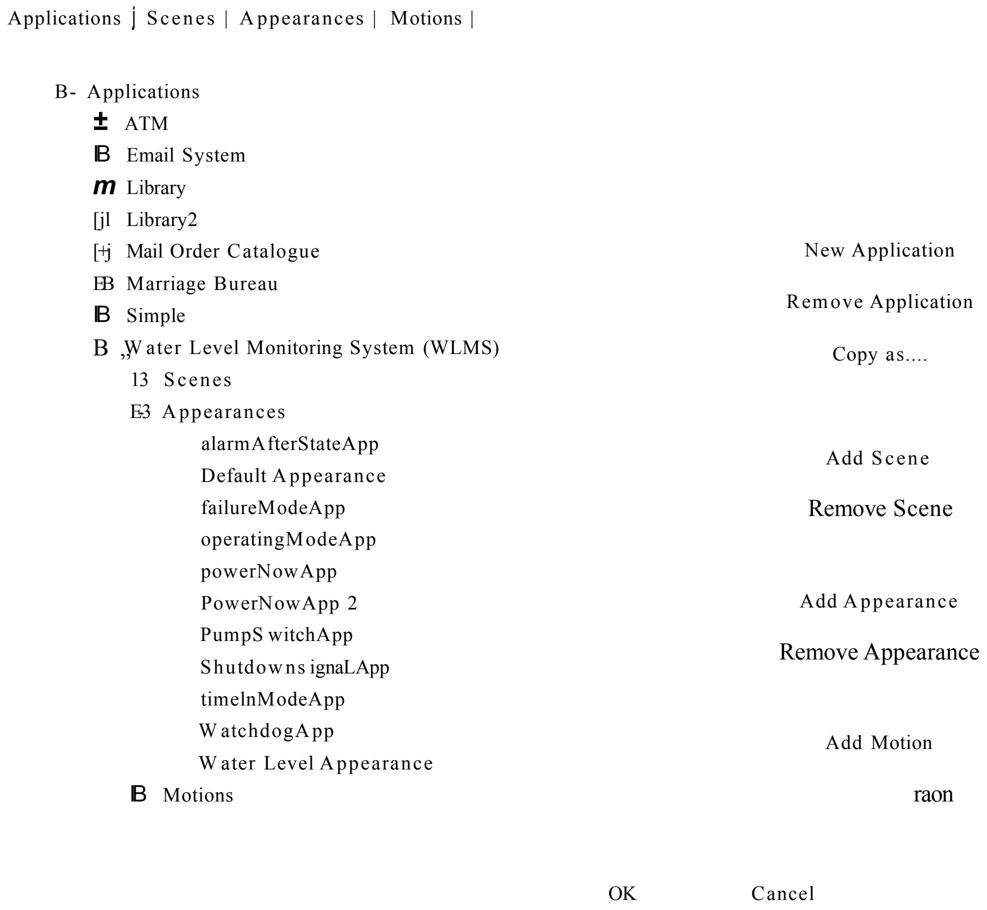


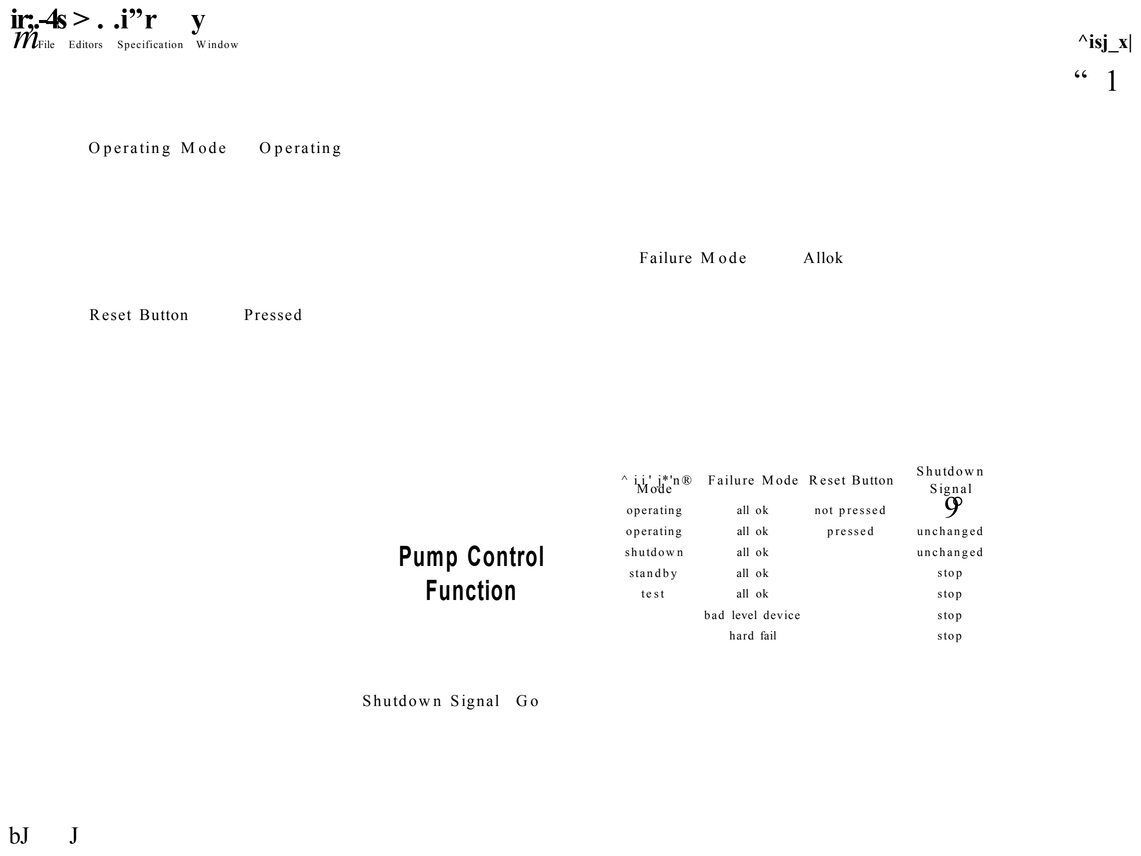
Figure 4.33. The contents of the WLMS Application Container.

Combining these lightweight constraints produces a practical approach for overcoming some of the problems that stem from the requirement to provide visually rich and highly flexible representation for developing visual prototypes but constrain the potential for applying these arbitrarily or inappropriately.

When the implementation of the scene has been completed, has been suitably associated with the PumpControl schema, and the schema has been executed, the visualisation shown in *Figure 4.34* results. This visualisation shows the inputs to the schema, and the single output that is the result of the processing conducted as the schema executes. To assist the validation process further, a small table containing the combinations and permutations of inputs and resulting outputs is also provided. This is simply a presentation of values as defined by the informal requirements of the WLMS. It can be used as an aid for any stakeholders to look-up the expected values and assess the accuracy of the execution of the schema. Such on-screen graphical aids, that augment the visualisation, can improve the effectiveness and efficiency of the overall validation process – judgements and decisions can be taken swiftly without having to resort to searching through paper-based specifications. Additionally, presenting such information can also form the basis for further discussion as to the appropriateness of the underlying conditions, expected values and results, and scenarios.

To conclude, this case study has demonstrated the development of a visual prototype for a substantial industrial strength application. Another aspect of the ViZ system, namely the constraint mechanisms that combine to facilitate the preservation of correspondence between the meaning of a visualisation and the meaning of the specification that is the focus of the visualisation effort, was also demonstrated.

The assurance mechanism, whilst lightweight, is significant. For systems that are not in the safety critical domain, for instance for the systems in the earlier Case Studies, the use of these mechanisms may not be considered appropriate or worthwhile. However, when validating systems in this particular domain, the assurance mechanism may become a vital part of the prototype construction process in an attempt to reduce all possibilities for error or ambiguity. Indeed, the corresponding preserving mechanism may be regarded in its own right as an important quality assurance technique.



Operating Mode Operating

Failure Mode Allok

Reset Button Pressed

Pump Control Function

^ i i' j*'n® Mode	Failure Mode	Reset Button	Shutdown Signal
operating	all ok	not pressed	9
operating	all ok	pressed	unchanged
shutdown	all ok		unchanged
standby	all ok		stop
test	all ok		stop
	bad level device		stop
	hard fail		stop

Shutdown Signal Go

Figure 4.34. The visualisation results for the execution of the PumpControl schema.

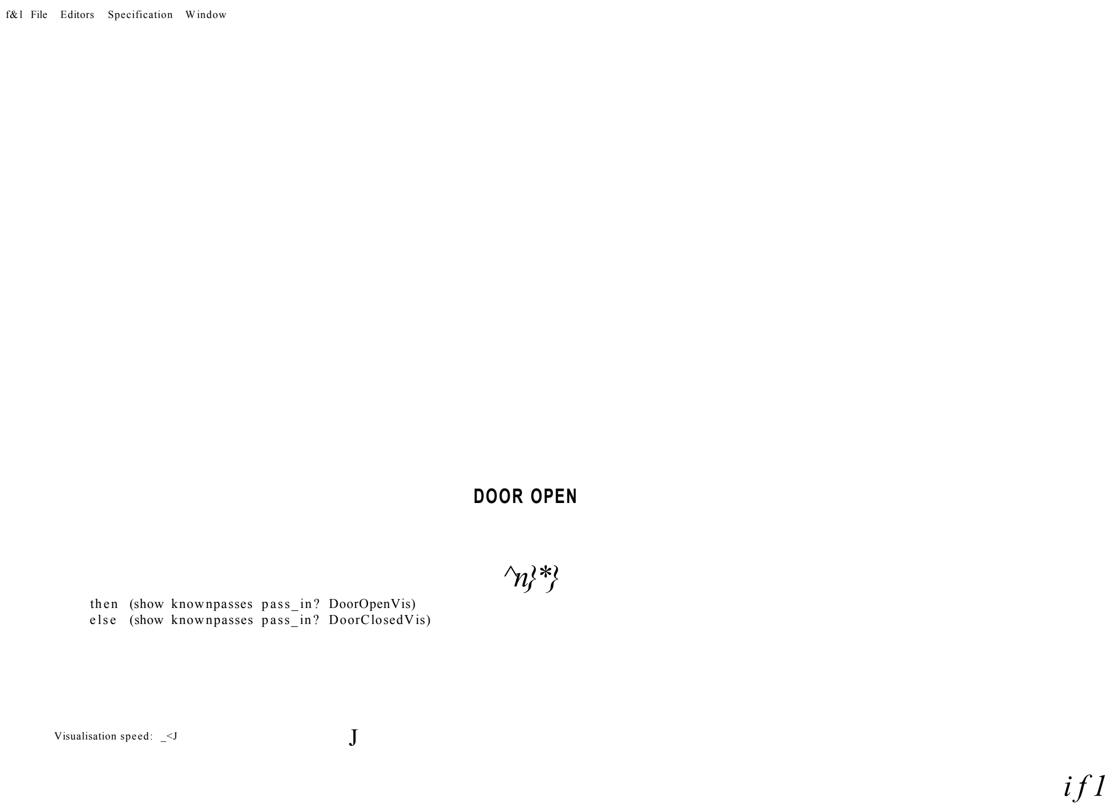
4.4 Case Study - A Security System

This case study is concerned with the development of a prototype that illustrates the behaviour of a secure entry system for a building.

4.4.1 Aims

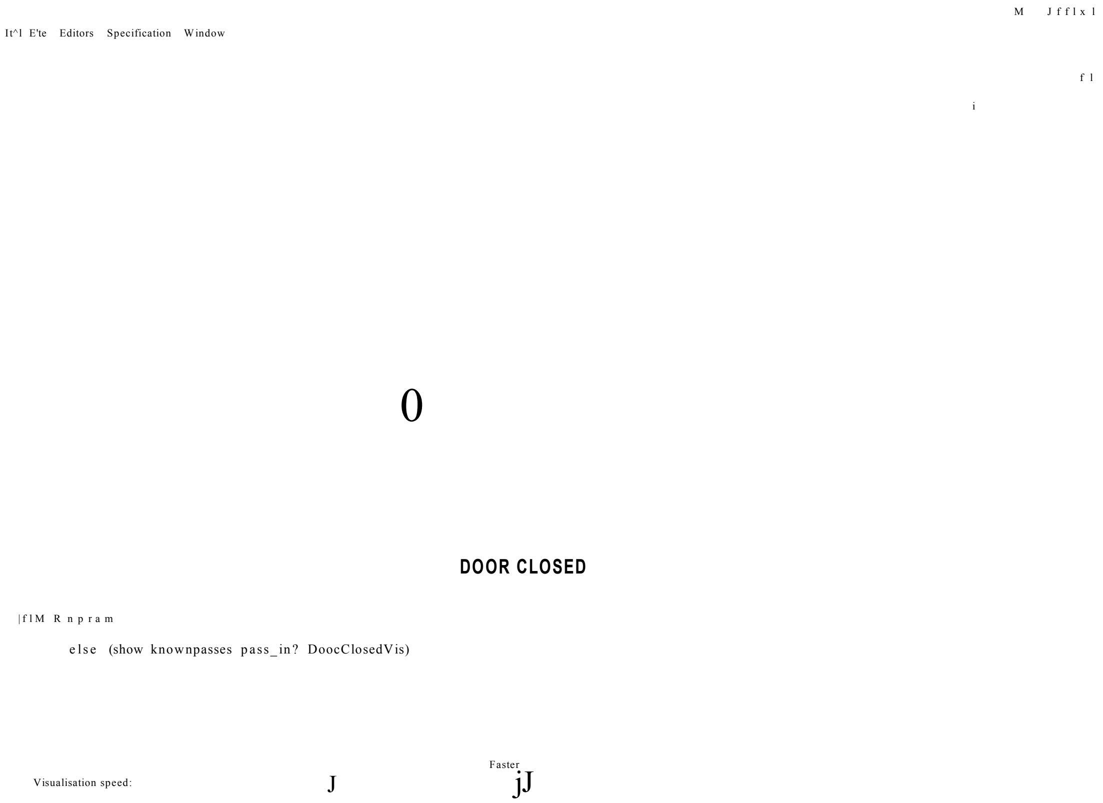
The aim of this case study is to demonstrate the effectiveness of ViZ in providing an effective platform with which to stimulate and engage users in the requirements validation process. To achieve this aim, this case study will involve real users in validating requirements for an upgrade to an existing system.

This case study will involve the development of two visual prototypes. The first will reflect the behaviour of the current system to provide a frame of reference for evaluating the requirements for the upgrade. The second will demonstrate the enhancements. This will enable the behaviour of the upgrade to be compared to the behaviour of the original system as well as providing a vehicle to explore the new features.



```
then (show knownpasses pass_in? DoorOpenVis)
else (show knownpasses pass_in? DoorClosedVis)
```

Figure 4.36. Results of executing the prototype for the successful door open scenario.



```
else (show knownpasses pass_in? DoocClosedVis)
```

Figure 4.37. Results of executing the prototype for unsuccessful door open.

The second visual prototype depicts the execution of several functions that pertain to the upgrade to the security system, namely entry, exit and the determination of the passes and persons inside the secure area.

To enter secure area with this system, the user exposes a known pass to the door sensor in the same way the as for the original system. However, the difference is that the pass identifiers are recorded. The second prototype models this through the “Enter” schema. Visually, the execution of this schema is depicted in a similar way to that of the first prototype, albeit with one difference. The visual metaphor for the first prototype employed representations that were simply constructed from geometric shapes to form a door-like presentation. The visual metaphor for the second prototype employs photographs of a door in open and closed states. The photographs include a person operating the door to add further realism. In addition, for the purposes of the developer, a display showing the passes recorded after the person has successfully entered the secure area is provided. Executing the “Enter” schema, using a pass identifier that is known to the system, produces a visualisation presenting an open door, as shown in *Figure 4.38.*

1- Itf l.X-i

fifd File £drtor$ Specification Window

3

_J 'S 3 2

I 4

```
then (show knownpasses pass_in? DoorOpenVis)
else (show knownpasses pass_in? DoorClosedVis)
```

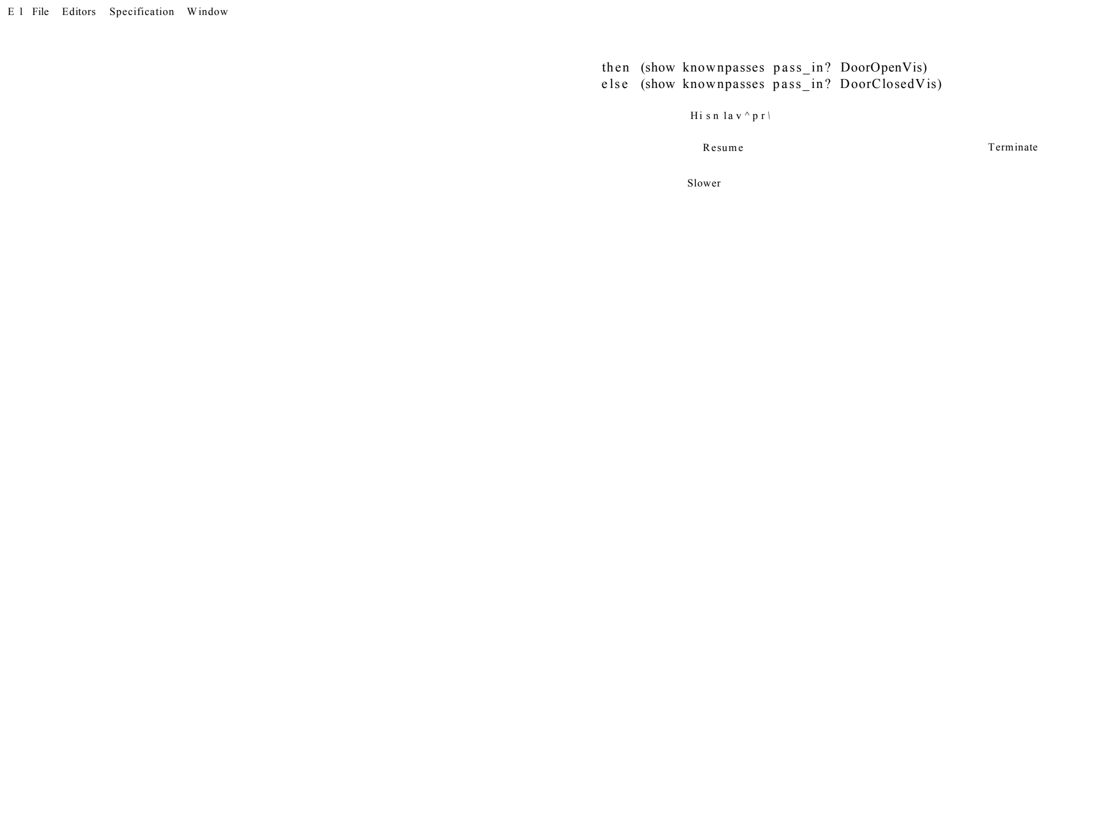
Resume

Visualisation speed: JIJ

Figure 4.38. Results ofsuccessfully executing the ‘Enter" schema.

The visualisation also presents the passes that are known to the system, as well as showing the pass identifier being used in the attempt to enter the secure area.

If the pass identifier is unknown to the system, then successful entry will not be granted. In this scenario, the pass being used to enter the secure area is not recorded. The visualisation that results presents a closed door. This can be seen in *Figure 4.39.*



```
then  (show knownpasses pass_in? DoorOpenVis)
else  (show knownpasses pass_in? DoorClosedVis)
```

Figure 4.39. Unsuccessful attempt to gain access to the secure area.

The function that models a person leaving the secure area removes the given pass from the set of pass identifiers that are recorded as being already inside. This is achieved after the person exiting exposes their pass to the sensor and the pass is recognised as being valid. The visualisation that is associated with this presents the door open or closed according to the result. The images are those used for entering the secure area.

The function to return the pass identifiers and the function to return the names of the personnel attributed to those pass identifiers do not have visualisations associated with them. Instead, the results of executing these functions is presented as text.

Evaluation Process

The prototype evaluation process will involve four separate activities. These are designed to provide the users with sufficient opportunity to familiarise themselves with the developer's understanding of the system (as represented by the specification) and opportunity to view, interact with and comment on the prototype.

The first activity is concerned with familiarising the users with the requirements and specification for the original system. This is performed to orient the users and provide them with a frame of reference for evaluating the requirements of the upgrade. It will be achieved by first allowing the users to view the informal specification of the original system and by the developers explaining the salient functions and system operations.

The second activity involves an examination of the prototype that models the requirements of the original system. The test cases devised in the earlier prototype development activities are used as the basis for this. The examination of this prototype is conducted by first explaining which scenario the prototype is being used to demonstrate and the conditions that prevail for the scenario to occur. The prototype is then executed with suitable test data. The resulting visualisations are then explained – their form, purpose and what they are depicting. This examination is conducted for the "Enter" function, and for the different scenarios that are associated with it. At this point, it is possible for the user to comment on the accuracy of the prototype with respect to the functionality of the original system. Even at this point, the developer may learn vital insights into the functionality of the system that is being upgraded. Also at this point, discussion may occur between the developer and user about the nature of this prototype and characteristics of its visual form.

The third activity entails the familiarisation of the user with the requirements specification for the upgrade. This is the first opportunity for validating the upgrade requirements. In this step, the developer will explain, on a function-by-function basis, their view of the requirements (as encapsulated by the informal specification). Since the users will be familiar with the underlying functions of the required system, they may be able to comment on the accuracy of the developer's understanding of it at this point.

The fourth and final step in the validation process involves the evaluation of the prototype of the upgrade. This is performed by executing the prototype in accordance with the scenarios and test data identified in the prototype development stage. Again, the developer will drive this step by 'walking' through the execution of each scenario. At this point, the users will be encouraged to articulate and discuss their thoughts and perceptions about what they see in the prototype. Any comments generated by the resulting discussions are to be recorded for future reference.

In practice, the next natural step would be to engage in the refinement of the prototype in response to the feedback generated from the evaluation process. In this case study this will not take place. Instead, and as stated previously, the focus is placed upon the user's involvement and the reaction and comments that emerge from the evaluation process.

Having described the requirements of the system, the form of the visual prototypes and the nature of the evaluation process, the next section will describe the results of enacting the evaluation process and report on the comments, feedback and issues that were raised.

4.4.4 Results

Undertaking the process elaborated in the previous section, and with the involvement of three prospective users of the security system, a number of discussions emerged. The users had wide experience with the functioning of the existing system as they used the door security system each time they entered their work environment.

In these discussions that emerged, a variety of points were raised about the system, the environment in which the system will operate and a number of factors that were not considered when the requirements specification was initially developed. This section will elaborate upon these discussions and describe the comments and feedback that resulted.

The discussions and the points raised can be partitioned into two categories. The first category includes the discussions that pertained to the prototype of the existing system. The second includes those that pertained to the prototype of the upgrade.

Discussions about the First Prototype

The users were first given a written specification explaining in plain English the functions of the existing system. After reading this, the users stated they were familiar with the functionality as described.

The examination of the first prototype's execution, and in particular in response to the presentation of the enactment of the successful door open scenario, lead to some fruitful dialogue about the original system.

The first important point that was raised was regarding the form of the prototype. Although the users did understand the system's requirements, they users did suggest that they would have preferred to see a more decomposed and sequential set of visualisations denoting the act of users exposing their passes, the door responding accordingly and the reasons for the door being either opened or remaining closed, instead of the visualisation of just the resulting open or closed doors. The visualisations used in the prototype were, they commented, sufficient and enabled them to see clearly the result of executing a scenario, but to add more realism and provide a more 'visual explanation' of what the user was supposed to be doing, a visualisations of the step-by-step actions of a user would have been preferable.

An important issue was raised with regard to the visualisation's ability to depict a user entering or exiting the secure area. The visualisation did not provide any visual clue to the direction of the user.

In addition, one user queried the application of this prototype. They wanted to know if this prototype specifically modelled their own door security system or whether it was a generic prototype that could be used to depict any such system. A discussion surrounding the purpose of the prototype then followed to clarify this matter.

At this point the discussion moved to the nature of the practicalities of the existing security system, and especially, the issue of a user opening the door with their pass and allowing extra people to enter without passes (for instance, if the user holds the door open). The prototype did not address this situation.

After the users appeared suitably familiarised with the prototype of the existing system, the evaluation process progressed to the examination of the prototype of the upgrade.

Discussions about the Second Prototype

The discussions that occurred during and after the presentation of the prototype's execution were useful and revealed a number of points that were not considered as the prototype of the upgrade was being developed.

The evaluation began by presenting the users with the informal requirements for the upgrade. The users read these and appeared eager to view the prototype that reflected them. In this prototype, photographs of the actual door that they are familiar with were used as the basis for the visual representations. This presented a slight difference, and the users were notified of this in advance.

Next, the scenarios identified in the earlier prototype development stages were enacted by executing the prototype with the appropriate test data in turn. Each scenario was explained, and the preconditions required for that scenario to arise were also described.

The first scenario chosen was that of an attempt to enter the secure area with a valid pass. Upon viewing the prototype the users repeated their comments about their preference of a sequence of visualisations depicting the actions a user must undertake to operate the system, instead of merely a single visualisation depicting the result. The discussion lead to additional requirements surfacing, not least the requirement to limit access to personnel with valid passes to specific times or dates (e.g. to restrict access in holidays or times outside of normal office hours). These requirements were not part of the original specification.

Another important issue was then raised. This involved the entry of visitors to the secure area. The users stated that this might be addressed by issuing temporary passes to visitors, but the practicalities of this needed clarifying (such as who would issue the passes, when would the visitors return the passes, etc).

The evaluation then progressed to the function for exiting the secure area. After viewing the execution of the prototype for the scenario that entailed a user with a valid pass attempting to exit the secure area, the users posed two important 'what if' style questions. The first question was based upon a scenario in which a non-valid user had slipped through security system (they had been allowed in by a valid user, for instance). What would happen if the person did not have a valid pass to exit, and no valid users were already in the secure area? The conclusion was that the person would not be able to exit. Second, and coupled to the first question, what if an emergency situation arose when such a person was inside the secure area and potentially could not exit? The conclusion was that the person would be in a dangerous, and potentially life-threatening situation.

This discussion then moved to the accommodation of emergency situations, and the potential of connecting the door security system to the fire alarm: if the fire alarm sounds, then the door locking mechanism should be instructed to unlock the door. Again, the prototype did not model this function.

Following this, the discussion returned back to issues surrounding a person exiting the secure area. The users reflected upon the complexities and general practicalities of an easy-to-use security system, and realised that some important decisions about the system and the environment in which it would operate would be required before a more complete implementation could take place.

The last system functions to be presented to the user what those concerned with returning the identifiers of the passes recorded as being inside the secure area and the names of those personnel associated with those passes. This, surprisingly, generated a discussion about legislation that would affect the security system. A legal requirement of any organisation applying software systems in UK is the Data Protection Act that prescribes how data can be handled by an organisation and how it may be divulged to users. This is pertinent, since Sheffield Hallam University (the commissioners of the system) would need to record personal details about the users of the security system, such as their names and potentially staff numbers, etc. A situation might also be envisaged whereby other details such as disabilities might be recorded – recording these might be useful in emergency situations for example. A building administrator might be able to see if any personnel with wheelchairs were inside the secure area if a fire broke out. The users pointed out that recording this type of information within the system does have Data Protection Act implications.

4.4.5 Observations and Conclusion

Having conducted a case study where real users are engaged in the validation process, a number of observations were made.

First, with regard to the requirements of the security system, much was revealed about additional features during the discussions that emerged. Several features that the developers did not include in the requirements specification for the upgrade was revealed. The discovery of these emergent properties was a direct product of the dialogue that took place.

Secondly, from the perspective of the utility of the ViZ system, the visual prototype provided the necessary vehicle on which the dialogue could be based. Without the prototype, the detailed discussions about the properties of the system(s) may not have occurred. All the users stated that the visual format of the requirements painted a largely accurate picture of the functionality of both the existing system and the upgrade, and that the prototype enabled them to see how the system would react to a given situation. The users also commented on how well the prototype accommodated and presented both the successful and unsuccessful scenarios.

It is also worthwhile to comment on the cost (in terms of effort and time) to develop the visualisations. The ViZ system's visual component repository enables the reuse of the visualisations, offering an effective reduction in the time needed to construct visual prototypes. In this case study, the functions being considered could be depicted by very similar visualisations, specifically an image of the security door open or closed. Once these visual representations were developed, they were applied to the Open Door and Closed Door functions very swiftly, with very little effort. Indeed, they could be applied to depict further scenarios with an equality small effort, resulting in a scaling of the visual prototype with relatively low development costs.

Lastly, looking from the viewpoint of the validating the research hypothesis (that visualisation is beneficial to the requirements validation process), it can be seen that this particular requirements validation project has been successful. It has facilitated dialogue, encouraged interaction between users and developers and resulted in requirements that were not originally thought of by the developers. The success, we can argue, is attributed to the facilities offered by the ViZ system.

4.5 Summary

This chapter has demonstrated the application of the ViZ system. It has shown the characteristics and capabilities of both the VIZ method and the corresponding software toolset. Three case studies were used to facilitate this demonstration. The first case study described the enaction of the ViZ method and how the functions of the toolset are used in close correspondence to develop visual prototypes. The second case study showed how the ViZ method scales to accommodate large systems. It showed that the method can be applied without an increase in complexity to develop visual prototypes of systems that possess significant quantities of functions/schemas. The third case study demonstrated both the application of the ViZ method to another significant system and how the correspondence preserving mechanisms in the system assist in preventing the inappropriate application of visual representations. The fourth and final case study demonstrated the effectiveness of the ViZ system in providing a platform for involving the user in the requirements validation process.

The final chapter provides an evaluation of the work as a whole against the original research objectives that were defined in *Chapter 1*. It augments this with a discussion as to the issues that have emerged as a consequence of undertaking the work. Finally, a description about the possibilities for future work is offered, along with some general remarks about the work.

Chapter 5
Critical Evaluation and Conclusion

This thesis has described research into the development of a novel approach to visualising prototype execution behaviour in the context of requirements engineering. The work was motivated by the recognition of the difficulties of presenting prototype behaviour using traditional representations during the validation of software and system requirements. Subsequently, the thesis identified desirable properties of an approach to visualise prototype execution by reviewing the current state of the art. Further, the thesis proceeded to describe the requirements, design and architecture of a system for visualising the execution of a specific requirements prototyping approach.

This final chapter presents an evaluation of the research and the conclusions that may be drawn from it. In particular, the achievements of the work are reviewed against the original research contributions and the initial research objective. Further, a discussion is presented concerning several issues that emerged as a consequence of conducting the research. Finally, the possibilities for future improvements of the ViZ system are identified in order to provide a basis for the continuation of the work, along with some general comments and conclusions concerning the work.

5.1 Thesis Contributions Revisited

The thesis has made a direct contribution to the field of requirements engineering, and more generally, to the discipline of software engineering. However, three specific contributions have been made. These were:

1. A review of the state of the art, evaluating and critiquing existing systems for visualising prototype execution.
2. The development of a system to facilitate visualisation of prototype execution.
3. A demonstration of the effectiveness of the system via the use of a set of case studies.

This section offers an evaluation of each of these contributions and establishes how well the research has succeeded in fulfilling these.

5.1.1 Survey of the State of the Art

The first contribution, the survey of the state of the art, was presented in *Chapter 2*. The survey was based upon a literature review and critique of several existing prototype execution visualisation systems. It was conducted by identifying the characteristics of the systems and partitioning these in accordance with a novel taxonomy. The taxonomy was defined using criteria established from the preceding discussions that established the background of motivation for the research (*Sections 2.2.1 and 2.2.2*). A critique of each system was offered and their strengths and weaknesses were discussed [Parry95]. This particular work preceded more recent and related research in reviewing and developing systems to facilitate requirements visualisation [Dulac02].

The objectives of conducting the survey were twofold. The first was to provide an overview of the current state of the art. The second was to establish a baseline for the subsequent development of an alternative and enhanced prototype execution visualisation approach. Combining and generalising the shortcomings identified in the reviewed systems achieved this. In this respect, the objectives of the survey were fulfilled, and it is believed that this contribution was successful and that it played an important part in the overall research.

5.1.2 Development of a System to Visualise Prototype Execution

The second contribution involved the development of a system to visualise prototype execution with the goal of presenting this behaviour in a customer-oriented manner. This contribution led to the development of the ViZ system. It is necessary therefore to evaluate the success of the ViZ system in fulfilling this goal. This evaluation will be performed in three stages. The first will offer an evaluation of the ViZ system with respect to its original requirements (as described in *Section 3.1.2*). The second will provide a more abstract assessment of the worth of the system in light of factors that were deemed critical to the success of such systems (as detailed in *Section 2.2.1*). The third stage will conduct a specific review of the ViZ system in the same manner as the survey of existing work. This will use the same taxonomy as that used for conducting the survey, as described in *Section 2.3*.

Evaluating the ViZ System against its Original Requirements

Evaluating the system against the requirements originally defined for it provides an assessment as to whether these requirements were fulfilled. To recap from *Section 3.1.2*, the requirements for the ViZ system were partitioned into three sections: visualisation provision, software integration and process support. The complete set of requirements are summarised in *Table 5.1.*

Requirements for Visualisation Provision

- Provide a range of visual cues
- Enable visual cues to be combined to produce visualisation objects to represent individual execution artefacts and contextual information.
- Enable visualisation objects to be combined to produce complete scenes.
- Facilities for the creation, storage and modification of visual representations
- Facilitate reuse
- Promote correspondence/integrity between visual representations and specifications

Requirements for Software Integration

- Provide separate tool support for visualisation facilities
- Facilitate tool inter-operation. This in turn requires communication support between the two systems
- Enable a minimal and efficient mechanism for associating visualisation details with specifications.

Requirements for Process Support

- Guide users through the activities inherent in developing visual prototypes
- Need to integrate activities of existing RealiZe approach with those responsible for visualisation development

Table 5.1. Summary of the requirements of the ViZ system.

In terms of visualisation provision, the ViZ software tool was designed specifically to provide a range of visual cue types and enable these to be combined to construct visualisation objects. This can be observed in operation in all three case studies.

In addition, the first case study was used as a vehicle for demonstrating how the ViZ tools accommodate the visualisation hierarchy. Providing this hierarchy was described as one of the challenges in developing prototype execution visualisation systems in *Section 2.2.3*. These tools provide a comprehensive environment for manipulating visual cues into visualisation objects (through the Appearance Editor) and subsequently arranging these into scenes (via the Scene Editor) that can be used to visualise a complete expression in a specification. This is augmented with tool support for specifying animation aspects of a visualisation through the dynamic-component editor.

The ViZ process also contributes significantly to visualisation provision. Direct support for visualisation development is provided through the scenario–based activities prescribed by the process.

In addition to creating and modifying visual representations, the visualisation repository provides a useful facility for their storage and their reuse. After creation with the ViZ editing tools, the representations are placed in a suitable container within the repository and can be recalled for further editing or application to a validation project. Again, the repository can be seen in action in the first case study.

The last requirement concerning visualisation provision, to provide a mechanism to promote integrity or correspondence between the meaning of visual representations and underlying specifications, is fulfilled by the mechanisms built into the visual repository and the visualisation editors. These were demonstrated in the third case study. They restrict the application of visual representations to only the areas for which they were deemed appropriate, or the types of execution artefact they were created to depict. These mechanisms are used in concert with the activities in the ViZ process that specify the classification and partitioning of visual representations at the time of their creation. Although the success of the assurance mechanism ultimately depends upon the users of the system being sensible, and that it is not an automatic mechanism, it can be seen that it offers an acceptable compromise between the flexibility and rigidity in terms of visualisation provision. The rationale for such a compromise was discussed in *Section 2.2.3*, where the need to retain flexibility over rigidity for the purpose of satisfying the visual needs of non-developers was noted.

In terms of software integration, the ViZ tools provide a number of important features that enable visualisation to be integrated with the ReaIiZe approach. The visualisation functionality, encapsulated in the ViZ tool and kept separate from the ZAL tool to facilitate the separation of concerns, satisfies the first requirement in this category.

Tool interoperation, the second requirement, is achieved by the innovative communication sub-system. The sub-system is innovative in that it draws upon lightweight technical features and simple principles and combines these to form the central aspect of the ViZ system. Without it, the ViZ system would not be able to visualise the execution of ZAL expressions directly. The sub-system is implemented through an underlying transport mechanism (based on a client-server, shared memory mechanism) augmented with an appropriate protocol to enable the separate tools to communicate. The communication sub-system is more than adequate for providing the required degree of tool interoperation.

The last requirement in this category specified the need for a minimal and efficient means of associating visualisation definitions with specifications, so at run-time, the definitions can be interpreted and applied to the respective execution artefacts. This requirement was considered throughout the design of the toolset and was implemented as both an addition to the syntax of the ZAL specification and a novel software architecture responsible for parsing and interpreting these syntax enhancements. At run time, the visualisation definitions are intercepted and the ZAL expressions are sent to the ZAL execution for processing. The results are then received and passed to the Visualisation Engine for rendering. The syntax enhancements and parsing mechanism were designed to provide a separation of the necessary details by making use of the principle of data hiding. By encapsulating the data that defines a visual representation into a visualisation object and then packaging several of these into scenes, and by referring to scenes by a unique name, a simple mechanism of association was established. This entailed simply implanting the scene identifiers into the specification. The parsing mechanism is then responsible for decoding the given scene and applying the visualisation objects contained within it. By attributing more of the work to software, a minimal approach to associating visualisation details to specification was defined. This particular aspect of the system is demonstrated in all three case studies.

The third category, process support, is fulfilled by the innovative ViZ Process. This central component of the system provides the necessary guidance, missing in many other systems for visualising prototype execution, for developers and other stakeholders. Importantly, the process prescribes the steps needed to develop visual prototypes in an effective and repeatable manner. From repeatability stems the increased likelihood that success, in terms of developing a visual prototype that is appropriate at portraying the requirements, will ensue. Experience in applying the system can be developed over time, leading to maturity, potential improvements and optimisations in the way the system is used. This is in contrast to the alternative: an ad-hoc situation occurring each time a visual prototype is developed.

At a more detailed level, the ViZ process provides integration with the RealiZe process. Indeed, the existing RealiZe approach was the starting point for developing the ViZ process. The ViZ process commences at the point when a ZAL specification exists and uses the ZAL specification as the primary input. To this end, the ViZ system fulfils this particular requirement.

Evaluating the ViZ System against the Critical Factors

Evaluating the system against the critical factors that were deemed necessary for such systems provides a more abstract review of the overall abilities of the system. The critical factors that were described in *Section 2.2.1* can be summarised as support, presentation, developer attitudes, development costs, and differences between developers and other stakeholders. The ViZ system will be evaluated against each of these.

The first critical factor concerns the support available for presenting prototype execution. In particular this factor stated that prototype execution behaviour should be presented in terms that non-technical stakeholders can understand. The ViZ system has been designed specifically to accommodate this. It has been the fundamental motivating force behind the development of the system. Overall, the system provides a comprehensive environment and process for creating and manipulating visual representations, and subsequently for associating these with a specification so that they can be rendered in real-time and in concert with the execution of the prototype. It is regarded therefore that this critical factor is sufficiently addressed.

The second critical factor, presentation factors, concerned the nature of the visual representations used to depict execution behaviour. It stipulated the necessity of providing 'rich' representations consisting of a wide variety of visual cues and the freedom to combine these into expressive visualisations. In addition, it also stated the need to facilitate correspondence and integrity between the meaning of a visual representation and the meaning of a specification it is used to depict.

By facilitating the importation of representations directly from the customer's own universe of discourse, and subsequently enabling their application to a prototype's execution, the ViZ system fulfils the requirement of offering visually rich representations. This is the innovative aspect of the system. The system does not restrict the developer or visualisation designer to using images of a particular type, or representations that are computed or generated automatically by the system. Therefore, the system overcomes a fundamental issue that was identified as being problematic in existing prototype execution visualisation systems.

In terms of correspondence, the ViZ system provides the assurance mechanism that manifests itself at various levels in the ViZ system. The assurance mechanism was designed to provide a balance between flexibility and rigidity. On reflection however, there are two potential drawbacks with this approach. The first is that the mechanism is indeed a compromise – and compromise often requires certain aspects to be omitted so others can be accommodated. The second drawback is somewhat less profound, in that the mechanism provides only an assurance – and not a guarantee – that the representations are appropriate for the given specification. It would be possible, either through accident or deliberate misuse, that a representation could be certified wrongly. This could lead to the possibility that a representation could be used inappropriately, resulting in ambiguity arising in the validation process. This would be a dangerous situation. The drawbacks notwithstanding however, it is still argued that the facilities offered by the ViZ system culminate in the provision of an environment that addresses this particular critical factor.

The third critical factor is developer attitudes. This concerns the preferences of developers when producing and using prototypes. It was stated in Section *2.2.1* that developers prefer technical, concise notations, and the rationale behind this. As well as accommodating non-technical stakeholders, by enabling the depiction of prototype execution in manner they can comprehend, the ViZ system also provides a simple and concise method by which visualisations can be described and attributed to specifications. The design of this particular component in the ViZ system was carefully chosen to reflect the desires of developers – an overly complex approach, with a possibly different syntax would have been inappropriate, and would probably require unnecessary effort to use.

A particularly important concern, and one that is reflected by the fourth critical factor, is that of development costs. When employing prototypes, in any form, in a software development process, it is imperative that every effort should be made to reduce the resources and costs required. In terms of addressing this particular factor, the ViZ system offers a rudimentary reuse mechanism manifested through the visual component repository and the polymorphic characteristics of the visual representations. This facility also enables a visual prototype to scale without a corresponding scaling in development effort. However, the efficacy of this can be questioned, especially when new validation projects are considered. The usefulness of reuse obviously depends upon the availability of visual representations at the time they are required. If no such visual representations exist then they must at least be created, thus resulting in no savings of effort or time. However, this is a problem that affects the reuse paradigm in general and not the ViZ approach specifically.

Recognition of the fact that differences exist between developers and stakeholders is the focus the fifth critical factor. This factor has lead to the development of the ViZ system. Indeed, the fundamental aim of the system has been to address these differences. Of importance however, is the need to engage the stakeholders in the validation process and enhance their experience in this respect. Visualising prototype behaviour using imagery and terminology with which they are familiar provides a large stepping-stone towards comprehension. From comprehension comes engagement, and then discussion about the requirements that the prototype represents. To this end, it is argued that the very existence of the ViZ system addresses this particular critical factor.

Review of the ViZ System

Evaluating the ViZ system using the approach used for surveying existing systems provides a means of classifying ViZ and comparing it to these systems. To this end, the same taxonomy will be employed as described in *Section 2.3*. The criteria that comprised this were described in detail in *Table 2.1*. The results of surveying the ViZ system are given in *Table 5.2*.

Criteria	Description	Extent
Application Domain	Unlimited. The ViZ system can be applied to an extensive range of domains and system types.	
Visual cues (the lowest level in the visualisation hierarchy)	A variety of visual cues are available, including geometric shapes, text, and the importation of photographic images.	✓✓✓
Visualisation objects (the intermediate level in the visualisation hierarchy)	Appearance components are provided that encapsulate the visual details necessary to depict an execution artefact or a single contextual detail. Dynamic components specify animation components.	✓✓✓
Scenes (the highest level in the visualisation hierarchy)	Scenes are provided as collections of appearance- and dynamic components. Used to depict/visualise complete ZAL expressions.	✓✓✓
Dynamism	Dynamism is provided through animation of appearances. Appearances are moved, in real time, around the screen when necessary.	✓✓
Flexibility	The Appearance-, Dynamic-, and Scene editors enable creation and modification of all necessary visual components using graphical user interfaces.	✓✓
Representing Relationships	No specific mechanisms are offered to depict relationships between visualisation objects. These must be created and displayed as contextual details by the developer/visualisation designer.	✓
Abstraction	No specific mechanisms are offered for automatically creating abstract views of the prototype execution. The developer/visualisation designer must create these through the existing visual apparatus.	
Correspondence	Correspondence between the meaning of a visual representation and specification is assured through the correspondence preserving mechanism. This requires the certification of visualisation components as being appropriate for a particular application, and the tools enforce this when they are being applied.	✓✓✓

Table 5.2*. Review and classification of the features of the ViZ system.*

5.1.3 Demonstration of the Effectiveness of the System

The third contribution of the research was to demonstrate the ViZ system. The aim was to provide a validation of the system by illustrating its capabilities and effectiveness. An evaluation of this particular contribution therefore centres upon determining the success of the validation effort.

The demonstration of the effectiveness of the system was performed through three case studies. These were documented in *Chapter 4*. The case studies also provided an opportunity for illustrating how the steps in the ViZ process are undertaken.

Evidence to support that the case studies were successful is twofold. First, although the case studies demonstrated the process of developing visual prototypes, the resulting prototypes were nonetheless important. Such prototypes showed the potential the system has for portraying requirements in a graphically oriented manner. Contrast these prototypes to the text-based prototypes created with the RealiZe system. The fundamental tenet of this thesis argues that the visual forms are indeed superior for communicating the meaning of the underlying requirements, and can be used to stimulate dialogue in ways that would not occur with the text-based counterparts. The case studies were successful in demonstrating this.

In addition, the process, as illustrated by the case studies, offers a means to arrive at and subsequently use visual prototypes in a repeatable manner. The case studies successfully demonstrated this repeatability. Each case study was carried out using the same systematic approach, in the style of a scientific experiment. This facilitated the demonstration of the similarities of applying the VIZ process to entirely different systems.

5.2 Evaluation Against the Original Research Objective

The original research objective was to develop an alternative prototype execution visualisation system. This was stated in *Chapter 1.* Through the individual research contributions, the problems identified with existing systems have been addressed. The properties that are otherwise desirable in such systems are also accommodated. The novel contribution made by the ViZ toolset and process combine to provide a comprehensive environment with which software developers and their customers can engage, with confidence, in the requirements validation process. To this end, the original research objective is satisfied.

5.3 Discussion

It has been established, and not surprisingly, that applying visualisation in the context of requirements validation is not a straightforward activity. Through applying ViZ to the case studies and when exercising its features during the testing phase of its development, several practical and technical issues have emerged that must be considered if the use of the ViZ system, and the application of visualisation to requirements engineering in general, is to be successful. This section discusses these issues.

First and foremost, it is observed that there is the need to devise visual representations that aptly portray the meaning of the requirements to which they are being applied. This may seem obvious. Although technical mechanisms may provide measures to address this fundamental issue, the responsibility ultimately lies with the developer or visualisation designer to create and apply appropriate visual representations.

There is also the need to address the visual requirements of the stakeholders. This is in relation to the above issue. When applying visualisation to any domain, meeting the visual requirements of the audience is important. In visualising prototype execution, this need is amplified. In many other domains where visualisation is applied, there may be perhaps one or two types of viewers, quite often with similar levels of technical skill and visual needs. However, in requirements validation, there may be many types of stakeholders, all with different vested interests in the proposed software system, and all with different levels of domain- or technical knowledge. This presents a problem, as this variety cannot always be accommodated through a single style of visual representation or appearance type. Addressing this issue could require both non-technical and technical mechanisms. Firstly, the intended audience's visual needs and capabilities must be identified before visualisation development takes place. Second, software developers or visualisation designers must not simply assume that their representations of a proposed software system are appropriate, and that alternative representations may be effective at communicating the behaviour of the system to the stakeholders. Additionally, the visualisation software must contain suitable features to permit multiple views. At present ViZ does not easily support multiple views of the same execution.

One particularly significant non-technical issue has emerged. During the development of visualisations for a variety of systems, it has become apparent that devising visual representations for some types of systems is more difficult than it is for others. The difficultly lies with the apparent degree of abstraction of the software system being considered. Some systems lend themselves quite easily to visualisation. These systems mostly have identifiable real-world counterparts, closely model real-world systems, or have tangible components (such as Abstract Interface Objects) with which a user can interact. These types of system may be regarded as being 'less abstract'. On the other hand, even though their requirements might be founded in the real world, some systems have no real-world counterpart; they might perform some kind of processing using complex algorithms whose internal operational details are hidden, or they may not have many inputs or outputs with which to easily gauge execution progress. These types of systems may be regarded as being 'more abstract', and present difficulties for visualisation development. Any solution certainly lies with the creative skill of the software developer or visualisation designer and their ability to invent visualisation schemes to portray the execution of such abstract systems.

Furthermore, it has been discovered that too much visualisation is potentially a bad thing. For most validation projects, it is simply not necessary or even desirable to visualise every detail of a prototype's execution. Although the ViZ system enables visualisation of potentially all execution artefacts, such power and freedoms must be used with restraint. Offering too many visual representations leads to 'information overload' on the part of the stakeholder, and this may lead to confusion, distraction, and incomprehensibility – negating the very benefits that the application of visualisation were intended to overcome. For example, during the case studies, we have found it unnecessary to visualise most system housekeeping operations (i.e. the rather mundane updates to the software system's state). These add little to the overall presentation of system behaviour non-technical stakeholders. However, we have found it necessary to provide visualisations for activities in which actors play major roles (or in other words, where users would directly be involved with the system). Importantly, the use of scenarios in the ViZ approach has provided us with a basis to establish where visualisation might be appropriate and best applied.

It is also important to exploit opportunities for reusing visual representations whenever they arise. Capitalising on reuse reduces the effort, time and cost required to develop visualisations. This is an important point; especially when software developers are under continuous pressure to reduce the 'time-to-market' and complete projects within allotted budgets. Additionally, visualisation reuse promotes visual consistency, whereby representations are associated repeatedly with the same execution artefacts. If a stakeholder becomes familiar with a particular association, their mental load will be reduced – they will not have to perform a translation between representation and artefact. If visual consistency is maintained within a validation project, the likelihood that comprehension will ensue is greatly increased.

Lastly, it is important to consider the issue of maintaining the effectiveness and utility of ViZ (and systems that facilitate visualisation of prototype execution behaviour in general) in light of continuous developments within the areas of visualisation and software engineering.

Developments in visualisation and graphical technologies, driven by research and market forces, are rapid and frequent. New hardware devices, software techniques and tool support emerge daily. Current developments include, for example:

- The advancement of real-time three-dimensional graphics, and especially making this technology more accessible to application developers through toolkits, libraries and application programmer interfaces (APIs).
- Professional-quality desktop video recording, editing and playback facilities, enabled by recent advances in processing power and storage capacities, that application developers can embrace and use in their software systems.
- Multimedia tools and environments with rich feature-sets and built-in scripting languages that can be used to assist in the process of producing graphical and multimedia assets (images, visualisations, animations, etc).

It is impossible that the design of a visualisation system such as ViZ could accommodate all future graphical developments in advance. Indeed, this was not in the original ViZ requirements brief. Instead, the emphasis should be placed on applying whatever technologies are appropriate to portray requirements in forms that are amenable to customer comprehension, as new technologies emerge.

Software engineering research and the evolution of practices in industry result in a constant stream of advances in the realm of developing software systems. These advancements occur across the entire spectrum of software engineering activities, ranging from principles and theory, to processes and methods, and on to techniques and tools. Of particular interest to this research are advancements in the areas of requirements engineering and software development processes. Recent advancements have seen the development and introduction of, for example:

- Agile methods, which are development processes that are responsive to changing situations (i.e. changes in requirements, changing in business strategies, and changes to environmental factors).
- Extreme programming, which is a development approach encompassing the entire lifecycle that combines novel software design and implementation methods.

Irrespective of the approach used to develop systems however, there is one fundamental activity that needs to be pursued before any development takes place, and this is capturing requirements. The flexibility of prototyping, in either throwaway or incremental forms, combined with its capacity to be incorporated into any software development lifecycle, makes it still the technique of choice when developers have to understand and capture unknown or partially known requirements. It can be argued therefore, that visualising prototype execution behaviour is also relevant and would support and enhance prototyping in any lifecycle context. Integrating the process of visualisation into different software development lifecycles then becomes the substantive issue.

5.4 Opportunities for Future Work

Through the development and application of the ViZ system, it has become apparent that several directions could be pursued in the future. These would not only address current shortcomings with the ViZ system, but also facilitate the further research that was beyond the original scope of this thesis. The work can be partitioned into two categories: 1) enhancements to the current ViZ system, and 2) further research into applying visualisation to prototyping in the context of requirements validation, using the ViZ system as a vehicle to achieve this. These will be elaborated in the following sections.

5.4.1 Enhancements to the ViZ System

This category of future work largely addresses technical issues that pertain to the current ViZ system. The aim is to provide enhancements or to address shortcomings that have been identified. It is envisaged that these additions could be implemented over the short- to medium term.

1. Embracing emerging technologies to that could benefit the presentation of prototype execution behaviour, as mentioned in the 'Discussion' in *Section 5.3*. This may be considered a significant enhancement to the ViZ system. The aim would be to develop a mechanism that enables new technologies to be used by the existing ViZ environment without the need to re-engineer the application each time a new technology became available. Such a mechanism is traditionally developed using a shared data model that specifies how data produced by one application can be interpreted and used by another. This enables an application to use the services of others by simply importing their products. Ultimately, it becomes a simple matter of interpreting data formats. For each data format being considered, an application requires an interpreter that specifies how the data shall be interpreted, imported and used. Each interpreter can be constructed in terms of a 'plug-in' that can be developed as new data formats become available. These could be developed in accordance with a standard that states how the plug-ins should be implemented. An appropriate software architecture that enables the plug-ins to be used by the application would also be required.

   In the ViZ system, this approach could be used to facilitate the importation of files created with emerging third-party multimedia or animation applications. The data in these files could then be applied by the ViZ application to visualising prototype execution. The ViZ system would not have to require knowledge of the external application in advance – instead, plug-ins could simply be developed when necessary. This would enable ViZ to take advantage of new technologies and tools when they became available.

2. In the review of the ViZ system, presented above in *Section 5.1.2*, two areas were identified as being deficient. Both of these concerned the visualisation features available for portraying relationships between visual objects and those available for depicting levels of abstraction. Presently, to depict relationships or levels of abstraction, the developer or visualisation designer must provide the appropriate visual constructions. An enhancement to the ViZ system would be required to

address this. The enhancement should comprise the necessary visual cues, augmented with appropriate user interface facilities to manipulate these.

3. The ViZ system should provide multiple views of the same prototype. Such a feature would enable different stakeholder types to view the requirements in ways that would be most appropriate for them. For example, managers may require a different style of visualisation to end-users – differences in levels of detail or appearances may be required. The ViZ system should provide a suitable mechanism to enable the simple switching of visualisation styles, turn visualisations on or off, and develop and associate different visual representations with the underlying specification. Such a mechanism would require modifications to both the ViZ toolset, further syntax enhancements to the ZAL notation, and refinements to the ViZ process.

5.4.2 Further Research in Applying Visualisation to Requirements Prototyping

This second category of future work concerns larger-scale research that might be undertaken over the longer-term.

1. One direction would be to apply the ViZ system to a greater number of validation projects in different application domains with the aim of conducting empirical studies into the effectiveness of applying visualisation to requirements validation. Such studies are required to indicate further research directions in applying visualisation to requirements engineering. Areas of study could include investigations into the cognitive dimensions that pertain to the way prototypes are comprehended so they can be portrayed more effectively in the validation process, or the most appropriate way of portraying particular application domains or system types. In each study, the ViZ system provides the means necessary produce visual prototypes.

2. Presently, the ViZ system is used at the latter end of the requirements inquiry cycle [Potts94]. The system relies upon the earlier activity of elicitation being undertaken to produce a formal specification on which the construction of a visual prototype can be based. One research opportunity would be to change this arrangement. The aim would be to devise an alternative process that enabled the

ViZ approach to be employed earlier in the requirements cycle to directly facilitate the elicitation and definition of a formal specification of software systems. Such a process would be a natural refinement of the ViZ approach and would further extend its usefulness in requirements engineering.

3. Over the longer term, another potentially useful research direction could entail the application of the visualisation technology encompassed within the ViZ system to a requirements engineering approach that does not rely upon formal specifications. The aim of this research would be to bring the benefits of prototype visualisation to the wider requirements engineering community – and to those who do not use formal specifications. Such a direction stems from the recognition that formal methods, for a number of reasons, have experienced only a limited adoption within the wider software engineering industry. However, dispensing with the mathematical basis for prototype development would necessitate the introduction of an approach that would still facilitate execution driven visualisation. A way forward may be found by devising an alternative prototype execution mechanism based upon simulation techniques, combined with visual programming [Myers92].

5.5 Concluding Remarks

Research in the discipline of computing is often initiated and conducted to address problems that emerge as a consequence of using existing tools, techniques or approaches. This research has been conducted to address a particularly challenging problem that has emerged in requirements engineering. The problem concerned the way prototype execution behaviour is communicated to customers when prototypes are developed using executable formal specifications. It is necessary for customers to comprehend their execution behaviour if they are to make reasoned decisions about the accuracy of the requirements liberated during the requirements process. The problem is such that prototyping approaches employing executable formal specifications often portray execution in ways that are more suited to developers than to customers. The research addressed the problem by developing a new and innovative prototyping environment, known as ViZ, which employs visualisation and graphical representations as the primary means of communicating prototype execution behaviour.

This thesis has documented the research. It has provided a survey of existing work in the field and identified deficiencies, described the requirements of the ViZ system in response to those deficiencies, then presented the design and architecture of the system. To validate the ViZ system, the thesis described its application to three case studies. In all, this thesis has clearly demonstrated that applying visualisation to requirements engineering has practical benefits, and that this application is justified.

It would also be prudent to mention potential advances in the field of visualising prototype execution. Conducting this research has revealed certain general trends and needs. It is envisaged that future developments will be based upon the mixing of software technologies with techniques from disciplines external to software engineering such as graphic design, human-factors and systems theory. Doing so would provide the necessary 'added value' that is required to enhance the underlying practice of visualising prototype execution. In addition, there still remains the need to augment both new and existing requirements engineering practices and tools with visualisation facilities. As a consequence, the development of new approaches should emerge that will enable a developer to perform requirements validation with greater confidence.

To conclude, it is hoped that through this research the practice of requirements engineering, and the wider field of software engineering will be better informed. Through persistent and practical research, and through the development of new approaches such as ViZ, the field of software engineering can be refined and advanced with the aim of addressing the problems that continue to face the software industry and associated disciplines.

References and Bibliography

[ACM03] *Association for Computing Machinery - Recommended Curriculum for Software Engineering*
http://sites.computer.org/ccse/
Accessed 30th July 2003

[Ae87] *Rapid Prototyping of Real-Time Software Using Petri-Nets*
Tadashi Ae, Reiji Aibara
1987 IEEE Workshop on Visual Languages
19-21 August, 1987, Linkoping, Sweden,
ISBN 91 7870 208 9, IEEE CS Press, p234-241

[Alford77] *A Requirements Engineering Methodology for Real-Time Processing Requirements*
Mack Alford
IEEE Transactions on Software Engineering.
January 1977, Vol SE-3, No 1, p60-9

[Andriole87] *Storyboard Prototyping for Requirements Verification*
S. J. Andriole
Large Scale Systems in Information and Design Technologies,
Vol 12, No 3, p231-247, 1987.

[Barrett94] *Process Visualisation - Getting the Vision Right is Key*
J. L. Barrett
Information Systems Management
Vol 11, No 2, 1994, p14-23

[Benford96] *Visualising and Populating the Web: Collaborative Virtual Environments for Browsing, Searching and Inhabiting Webspace*
S. Benford, D. Snowdon, C. Brown, G. Reynard, R. Ingram
Proc JENC8 - 8th Joint-European Networking Conference JENC8.
1997, 123/1-9

[Bertrand97] *Grail/Kaos: A Tool for Goal-Directed Requirement Analysis*
P. Bertrand, R. Darimont, J. Declercq, E. Delor, P. Massonet,
A. van Lamsweerde
Genie-Logiciel. Dec. 1997; (46): 139-44
EC2 & Development

[Blumofe88] *Executing Real-Time Structured Analysis Specifications*
R. Blumofe
ACM Sigsoft Software Engineering Notes
Vol 13, No 3, July 1988, p32-40

[Bocker86] *The Enhancement of Understanding Through Visual Representations*
Heinz-Dieter Bocker, Gerhard Fischer, Helga Nieper
Proceedings of the CHI'86 Conference - Human Factors In Computing Systems III
13-17 April 1986, Boston, USA, Edited by Marilyn Mantei, Peter Orbeton,
ISBN 0-444-70052-8, p44-50

[Boehm76] *Software Engineering*
Barry W. Boehm
IEEE Transactions on Computers
Vol 25, No 12, p1226-1241, December 1976

[Boehm81] *Software Engineering Economics*
Barry W. Boehm
Prentice Hall, Englewood Cliffs, NJ, 1981

[Boehm84] *Verifying and Validating Software Requirements and Design Specifications*
Barry W. Boehm
IEEE Software
January 1984, p75-88

[Boehm88] *A Spiral Model of Software Development and Enhancement*
Barry W. Boehm
IEEE Computer
May, 1988, p61-72

[Booch99] *The Unified Modelling Language User Guide*
Grady Booch, James Rumbaugh, Ivar Jacobson
Addison Wesley, 1999

[Brooks87] *No Silver Bullet - Essence and Accidents of Software Engineering*
Frederick P. Brooks, Jr.
IEEE Computer
Vol 20, No 4, April 1987, p10-19

[Buckberry99] *An Editor and Transformation System for a Z Animation Case Tool*
Graham R. Buckberry
PhD Thesis
Sheffield Hallam University, August 1999

[Camara94] *Pictorial Modelling of Dynamic Systems*
Antonio S. Camara, Francisco C. Ferreira, Edmundo Nobre, Jose E. Fialho
Systems Dynamics Review
Vol 10, No 4, 1994, p361-373

[Carey90] *Prototyping: Alternative Systems Development Methodology*
J. M. Carey
Information and Software Technology
Vol 32, No 2, March 1990, p119-126

[Carr95] *Experiments in Process Interface Descriptions, Visualisations and Analyses*
David C. Carr, Ashok Dandekar, Dewayne E. Perry
Software Process Technology 4th European Workshop
April 1995, p119-137, ISBN 3540592059

[Chen76] *The Entity Relationship Model – Towards a Unified View of Data*
P. Chen
ACM Transactions on Database Systems
Vol 1, No 1, p9-36, 1976

[Cohen86] *The Specification of Complex Systems*
B. Cohen, W. T. Harwood, M. I. Jackson
Addison Wesley, 1986, ISBN 020114400x

[Combs92] Large Text Database Visualisation
N. Combs
Advances in Classification Research
Vol 3, Proceedings 3rd ASIS SIG/CR Classification Research Workings
25 Oct 1992, Pittsburgh, PA, USA, ISBN 0 938734 792 p1-14.

[Cooling94] *Making Formal Specifications Accessible Through the Use of Animation Prototyping*
J. E. Cooling, T. S. Hughes
Microprocessors and Microsystems
Vol 18, No 7, September 1994, p385-392

[Cox89] *PROGRAPH: A Step Towards Liberating Programming from Textual Conditioning*
P. T. Cox, F. R. Giles, T. Pietrzykowski
1989 IEEE Workshop on Visual Languages
October 4-6, 1989, Rome, Italy, IEEE Computer Society Press
ISBN 0-8186-2002-1

[Daly77] *Management of Software Development*
E. B. Daly
IEEE Transactions on Software Engineering
May 1977; Vol SE-3, No 3, p229-42

[Davis88] *A Comparison of Techniques for the Specification of External System Behaviour*
Alan M. Davis
Communications of the ACM
Vol 31, No 9, September 1988, p1098-1115

[Davis92] *Operational Prototyping: A New Development Approach*
Alan M. Davis
IEEE Software
September 1992

[Davis93] *Software Requirements: Objects, Functions and States*
Alan M. Davis
Prentice Hall
Englewood Cliffs, 1993

[DeMarco78] *Structured Analysis and System Specification*
Tom DeMarco
Yourden Press, 1982

[Diaz-Gonzales87] *Envisager: A Visual, Object-Oriented Specification Environment for Real-Time Systems*
Jose P. Diaz-Gonzales, Joseph E. Urban
Proceedings 4th Int. Workshop on Software Specification and Design
1987, p13-20

[Dobson94] *Learning Through and by Visualisation: A Case of Inter-Media Translation*
Michael W. Dobson
IFIP Transactions A - Computer Science and Technology
Vol 48, 1994, p77-94

[Domik93] *Scientific Visualisation*
G.O. Domik
Educational Multimedia & Hypermedia Annual Conference –
Proceedings of ED-MEDIA '93 - World Conf. on Education Media and Hypermedia
23-26 June 1993, Orlando, USA, p153-160, ISBN 1 88009 406 1

[Dorfman90] *System and Software Requirements Engineering*
Merlin Dorfman
in System & Software Requirements Engineering
Edited by Richard H. Thayer, Merlin Dorfman,
IEEE Computer Society Press, 1990, ISBN 0 8186 8921 8, p4-16

[Dorfman97] *Requirements Engineering*
Merlin Dorfman
in Requirements Engineering (2nd Ed)
Edited by Richard H. Thayer, Merlin Dorfman,
IEEE Computer Society Press, 1997, p7-22

[Duisberg87] *Visual Programming of Program Visualisations*
Robert Anthony Duisberg
1987 IEEE Workshop on Visual Languages
August 19-21, 1987, Linkoping, Sweden, IEEE Computer Society Press
ISBN 91 7870 208

[Dulac02] *On the Use of Visualization in Formal Requirements Specification*
Nicolas Dulac, Thomas Viguier, Nancy Leveson, Margaret-Ann Storey
IEEE Joint Int. Conference on Requirements Engineering (RE'02)
Essen, Germany, 9-13th September 2002

[Evans94] *Visualising Concurrent Z Specifications*
A. S. Evans
Z User Workshop - Proceedings 8th Z User Meeting
29-30 June 1994, Cambridge UK, Springer-Verlag, p269-281
ISBN 3 540 19884 9

[Faulk97] *Software Requirements: A Tutorial*
S. R. Faulk
in Requirements Engineering (2nd Ed)
Edited by Richard H. Thayer, Merlin Dorfman,
IEEE Computer Society Press, 1997, p128-149

[Feijs98] *3D Visualization of Software Architectures*
L. Feijs, R. De-Jong
Communications of the ACM. December. 1998
Vol 41, No 12, p72-8

[Filipidou98] *Designing with Scenarios: A Critical Review of Current Research and Practice*
D. Filipidou
Requirements Engineering Journal
Volume 3, No 1, 1998, p1

[Folin92] *The Essentials of Scientific Visualization: Basic Tools and an Example*
S. E. Follin
Social-Science-Computer-Review, Fall 1992, Vol 10, No 3, p337-354

[Frenkel88] *The Art and Science of Visualizing Data*
Frenkel,-K.-A.
Communications of the ACM. February, 1988, Vol 31, No 2, p110-21

[Fraioli00] *Storyboarding 101*
James O. Fraioli
Michael Wiese Productions, ISBN 0 941188 25 6, 2000

[Fujii97] *Software Verification and Validation*
Roger U. Fujii, Dolores R. Wallace
in Software Engineering
Edited by Merlin Dorfman, Richard H. Thayer
IEEE Computer Society Press, 1997, p220-234

[Gane79] *Structured Systems Analysis: Tools and Techniques*
C. Gane and T. Sarson,
Prentice-Hall, 1979

[Gershon98] *Information Visualization*
N. Gershon, S. G. Eick, S. Card,
Interactions, Vol. 2, March-April, 1998

[Gibbs94] *Software's Chronic Crisis*
W. Wayt Gibbs
Scientific American
September 1994

[Goguen93] *Techniques for Requirements Elicitation*
Joseph Goguen, Charlotte Linde
Proceedings, Requirements Engineering '93
Edited by Stephen Fickas, Anthony Finkelstein,
IEEE Computer Society, 1993, p152-164

[Gomaa83] *The Impact of Rapid Prototyping on Specifying User Requirements*
Hassan Gomaa
ACM Sigsoft. Software Engineering Notes
Vol 8, No 2, April 1983, p17-28

[Gomaa86] *Prototypes - Keep Them or Throw Them Away?*
Hassan Gomaa
Prototyping: State of the Art Report.
Edited by M. Lipp, Oxford, UK, Pergamon Infotech Ltd, 1986
ISBN 008 034 0938

[Gomaa90] *The Impact of Prototyping on Software System Engineering*
Hassan Gomaa
System & Software Requirements Engineering
Edited by Richard H. Thayer, Merlin Dorfman,
IEEE Computer Society Press, 1990, ISBN 0 8186 8921 8

[Green91] *Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the `Match-Mismatch' Conjecture*
T. R. G. Green, M. Petre and R. K. E. Bellamy
Empirical Studies of Programming: Fourth Workshop
1991, New Brunswick, NJ
Ablex Publishing Corporation. p121-146

[Hagen00] *Scientific Visualization - Methods and Applications*
H. Hagen, A. Ebert, R.H. Van-Lengen, G. Scheuermann
Lecture Notes in Computer Science, Vol 2000 – "Informatics - 10 Years Back, 10 Years Ahead", p311-327
Editor R. Wilhelm
Springer Verlag

[Hardgrave93] *Prototyping Effects on the System Development Life Cycle: An Empirical Study*
Bill C. Hardgrave, E. Reed Doke, Neil E. Swanson
Journal of Computer information Systems
Vol 33, No 3, 1993, p14-19

[Hart99] *The Art of the Storyboard*
John Hart
Focal Press
ISBN 0 240 80329 9, 1999

[Hayes86] *Specification Case Studies*
I. J. Hayes
Prentice Hall, 1986, ISBN 0 13 826595 X

[Hayes89] *Specifications Are Not Necessarily Executable*
I. J. Hayes, C. B. Jones
Software Engineering Journal, November 1989, p330-338

[Hekmatpour88] *Software Prototyping: Formal Methods and VDM*
S. Hekmatpour, D. Ince
Addison Wesley, 1988

[Heymans98] *Scenario-Based Techniques for Supporting the Elaboration and the Validation of Formal Requirements*
M.D. Heymans, E. Dubois
Requirements Engineering Journal
Volume 3, No 3&4, 1998, p202

[Hooper82] *Scenario-Based Prototyping for Requirements Identification*
James W. Hooper, Pei Hsia
ACM Sigsoft Software Engineering Notes, 1982, Vol 7, No 5, p88-93

[Humphreys89] *Visual Cognition: Computational, Experimental and Neuropsychological Perspectives*
Glyn W Humphreys, Vicki Bruce
Pub: Hove Erlbaum., 1989
ISBN 0863771254

[Hung02] *Models of the Visual System*
Edited by George K. Hung, Kenneth J. Ciuffreda
Kluwer Academic/Plenum Publishers
ISBN 0306467151

[IEEE98a] *IEEE Recommended Practice for Software Requirements Specifications*
IEEE Std 830-1998, IEEE Inc. 1998

[IEEE98b] *IEEE Guide for Developing System Requirements Specifications*
IEEE Std 1233-1998, IEEE Inc. 1998

[Jackson93] *Formal Specification and Animation of a Water Level Monitoring System*
P. S. Jackson, P. A. Stokes
Technical Report INFO-0428
Atomic Energy Control Board, Ottawa Canada
March 1993

[Jensen81] *Coloured Petri Nets and the Invariant Method*
K. Jensen
Theoretical Computer Science
June 1981, Vol 14, No 3, p317-36

[Jones90] *Systematic Software Development Using VDM*
C. B. Jones
Prentice Hall, Englewood Cliffs, 1990

[Kotonya00] *Requirements Engineering, Processes and Techniques*
G. Kotonya, I. Sommerville
John Wiley, 1998

[Lalioti93] *Visualisation for Validation*
V. Lalioti, P. Loucopoulos
Lecture Notes in Computer Science 685
Edited by Colette Rolland, Francois Bodart, Corine Cauvet
Advanced Information Systems Engineering 5th Int. Conf. CAiSE'93,
p586-600, June 1993, Paris, France, Springer Verlag, ISBN 3 540 56777 1

[Lamsweerde01] *Goal Oriented Requirements Engineering: A Guided Tour*
A. van-Lamsweerde
Proceedings 5th IEEE International Symposium on Requirements Engineering
2000, p249-62

[Lamsweerde95] *Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt*
A. Van Lamsweerde, R. Darimont, P. Massonet
Proc 2nd International Symposium on Requirements Engineering (RE95)
York, England, 1995, p194-203

[Larkin87] *Why a Diagram is (Sometimes) Worth Ten Thousand Words*
Jill H. Larkin, Herbet A. Simon
Cognitive Science, Vol 11, 1987, p65-99

[Lohse94] *A Classification of Visual Representations*
Gerald L. Lohse, Kevin Biolsi, Neff Walker, Henry H. Rueter
Communications of the ACM
Vol 37, No 12, December 1994, p36-49

[Luqi88] *A Computer-Aided Prototyping System*
Luqi, Mohammad Ketabchi
IEEE Software
March 1988, p66-72

[Luqi89] *Software Evolution Through Rapid Prototyping*
Luqi
IEEE Computer
May 1989

[Mantey94] *A View of Visualisation - Its Origins, Development and New Directions*
P. E. Mantey
SPIE – Proc. of the Society of Photo-Optical Instrumentation Engineers
Vol 2178, No 22, 1994, p2-11

[Marca88] *SADT: Structured Analysis and Design Technique*
D. A Marca, C. L. McGowan
McGraw-Hill Co.
1988

[Margaria98] *Formal Methods and Customised Visualisation: A Fruitful Symbiosis*
T. Margaria, V. Braun
Lecture Notes in Computer Science, No. 1385 - "Services and Visualisation", p190-207, Pub: Springer Verlag, 1998

[Marr82] *Vision. A computational Investigation into the Human Representation and Processing of Visual Information*
David Marr
Freeman, 1982

[Martin93] *Visualisation for Telecommunications Network Planning*
J. C. Martin
IFIP Transactions B - Applications in technology
Vol 9, 1993, p327-334

[McConnell03] *Software Project Survival Guide*
Steve McConnell
Microsoft Press, ISBN: 0072850612

[Meyer85] *On Formalism in Specifications*
Bertrand Meyer
IEEE Software, January 1985, p6-26

[Morrey98] *A Toolset to Support the Construction and Animation of Formal Specifications*
Journal of Systems and Software
No 41, 1998, p147-160

[Myers88] *Automatic Data Visualisation For Novice Pascal Programmers*
Brad A. Myers, R. Chandhok, A. Sareen
IEEE Workshop on Visual Languages
October 10-12, 1988, Pittsburgh, PA, USA, p192-198
IEEE Computer Society Press

[Myers92] *Survey on User Interface Programming*
Brad A. Myers, Mary Beth Robson
CHI'92 Conference Proceedings. ACM Conference on Human Factors in Computing Systems
3-7 May 1992, Monterey, CA, USA, ACM Press, p195-202
ISBN 0 89791 513 5

[Myers02] *Exploring Visualization of Complex Telecommunications Systems Network Data*
M. Myers, R. Sterritt, E. P. Curran, Hongzhi-Song, A. E. Kamel,
K. Mellouli, P. Borne
2002 IEEE International Conference on Systems, Man and Cybernetics
Conference Proceedings Cat. No. 02CH37349, Vol.7
IEEE Computer Society Press

[Narayanan95] *Language Visualisation: Applications and Theoretical Foundations of a Primitive-Based Approach*
A. Narayanan, D. Manuel, L. Ford, D. Tallis, M. Yazdani
Artificial Intelligence Review
No 9, 1995, p215-235

[Naur69] *Software Engineering: Report on a Conference Sponsored by the NATO Science Commission*
P. Naur, B. Randell
Garmisch, Germany, 7-11 October 1968
Scientific Affairs Division, NATO, Brussels

[Ozcan98a] *Requirements Validation Based on the Visualisation of Executable Formal Specifications*
M. B. Ozcan, P. W. Parry, I. C. Morrey, J. Siddiqi
Proceedings COMPSAC 1998, Vienna, Austria, August 1998
IEEE Computer Society Press

[Ozcan98b] *Visualisation of Executable Formal Specifications for User Validation*
M. B. Ozcan, P. W. Parry, I. C. Morrey, J. Siddiqi
Lecture Notes in Computer Science, No. 1385 - "Services and Visualisation", p142-157, Pub: Springer Verlag, 1998

[Parnas69] *On The Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System*
David L. Parnas
Proc. 24th National ACM Conference, 1969, p379-385

[Parnas77] *The Use of Precise Specifications in the Development of Software*
David L. Parnas
Proceedings IFIP, Toronto, Canada, August 1977, p861-867

[Parry00] *Development of a Visual Requirements Validation Tool*
P. W. Parry, M. B. Ozcan
Procedings 2nd International Symposium on Constructing Software Engineering Tools (CoSET 2000), June 2000, Limerick, Ireland, p112-120

[Parry95] *The Application of Visualisation to Requirements Engineering*
P. W. Parry, M. B. Ozcan, J. Siddiqi
Proc. Conference on Software Engineering and its Applications
Paris, France, 1995, p699-701

[Paulk93] *Capability Maturity Model, Version 1.1*
M. C. Paulk, B. Curtis, M. B. Chrissis, C. V. Weber
IEEE Software, July 1993, Vol 10, No 4, p18-27

[Peterson81] *Petri Net Theory and the Modelling of Systems*
J. L. Peterson
Prentice Hall, Englewood Cliffs, 1981

[Petre90] *Where to Draw the Line With Text: Some Claims by Logic Designers About Graphics in Notation*
M. Petre, T.R.G. Green
Proceedings Human-Computer Interaction - Interact'90
Third International Conference on HJuman-Computer Interaction
27-31 August 1990, Cambridge, UK, North-Holland, ISBN 0 444 88817 9

[Petri62] *Kommunikationen mit Automaten*
C. A. Petri
PhD Thesis, University of Bonn, 1961
English Translation: Technical Report RADC-TR-65-377, Vol 1, Suppl 1
Applied Data Research, Princeton, NJ

[Pillet95] *PARAVER : A Tool to Visualize and Analyse Parallel Code*
V. Pillet, J. Labarta, T. Cortes, S. Girona
WoTUG-18, p17--31, Manchester, April 1995

[Potts94] *Inquiry-Based Requirements Analysis*
Colin Potts, K. Takahashi, A.I. Anton
IEEE-Software. March 1994,
Vol 11, No 2, p21-32

[Pressman97] *Software Engineering*
Roger S. Pressman
in Software Engineering
Edited by Merlin Dorfman, Richard H. Thayer
IEEE Computer Society Press, 1997, p220-234

[QAA03] *Subject Benchmark – Computing*
The Quality Assurance Agency for Higher Education
http://www.qaa.ac.uk/
Accessed 9th August 2003

[Rasure91] *An Integrated Data Flow Visual Language and Software Development Environment*
John R. Rasure, Carla S. Williams
Journal of Visual Languages and Computing
Vol 2, 1991, p217-246

[Regnell95] *Improving the Use Case Driven Approach to Requirements Engineering*
Bjorn Regnell, Kristofer Kimbler, Anders Wesslen
Proc. 2nd IEEE International Symposium on Requirements Engineering
March 27-29, 1995, York, England, p40-47.

[Reiss01] *Encoding Program Executions*
S. P. Reiss, M. Renieris
Proceedings 23rd International Conf. on Software-Engineering (ICSE)
2001, p221-30, IEEE Computer Society Press

[Remington97] *Computer-Human Interface Software Development Survey*
Robert J. Remington
in Software Engineering
Edited by Merlin Dorfman, Richard H. Thayer
IEEE Computer Society Press, 1997, p220-234

[Rolland98] *A Proposal for a Scenario Classification Framework*
C. Rolland, C. Ben Achour, C. Cauvet, J. Ralyte, A. Sutcliffe, N. Maiden, M. Jarke, P. Haumer, K. Pohl, E. Dubois, P. Heymans
Requirements Engineering Journal
Volume 3, No 1, 1998, p23

[Ross77] *Structured Analysis: A Language for Communicating Ideas*
D. Ross,
IEEE Transactions on Software Engineering
Vol 3, No 1, p16-34, January 1977

[Royce70] *Managing the Development of Large Software Systems: Concepts and Techniques*
Winston W. Royce
Western Electronic Show and Convention – WESCON
Aug. 25-28 1970, Los Angeles, p.~A/1-1-A/1-9

[Rumbaugh94] *Object-Oriented Modeling and Design*
J. Rumbaugh
Object Expo - Conference Proceedings
1994, p249-55
SIGS Publications, New York, NY, USA

[Scharer81] *Pinpointing Requirements*
Laura Scharer
Datamation, April 1981, p17-22

[SEI03] *Software Engineering Institute - Capability Maturity Models*
http://www.sei.cmu.edu/cmm/cmms/cmms.html
Accessed 9th August 2003

[Sekuler94] *Perception*
Robert Sekuler, Randolph Blake
McGraw Hill, 1994, ISBN 0070560854

[Shu89] *Visual Programming: Perspectives and Approaches*
N.C. Shu
IBM Systems Journal
Vol 28, No 4, p525-547, 1989

[SIGRAPH03] *ACM Special Interest Group on Graphics and Interactive Techniques*
http://www.siggraph.org/
Accessed 8th August 2003

[Sommerville97] *Requirements Engineering*
I. Sommerville, P Sawyer
John Wiley, 1997

[Spence01] *Information Visualisation*
Robert Spence
Addison Wesley, ACM Press, 2001

[Spice03] *Software Process Improvement and Capability Determination*
http://www.sqi.gu.edu.au/spice/
Accessed 13th August 2003

[Spivey92] *The Z Notation: A Reference Manual (2nd Ed)*
J. M. Spivey
Prentice Hall, 1992

[Spoehr82] *Visual Information Processing*
Kathryn T. Spoehr, Stephen W. Lehmkuhle
Oxford, Freeman, 1982
ISBN 0716713748

[Sprott97] *Scientific Visualization in Mathematics and Physics*
J. C. Sprott, C. A. Pickover
Interactions, January-February 1997, Vol 4, No 1, p78-9
ACM Press

[Standish98] *The Chaos Report*
The Standish Group. 1998
http://www.standishgroup.com
Accessed 31st July 2003

[Stasko92] *Understanding and Characterising Software Visualisation Systems*
John T. Stasko, Charles Patterson
Proceedings of the 1992 IEEE Workshop on Visual Languages
September 15-18, 1992, Seattle, Washington
IEEE Computer Society Press, ISBN 0 8186 3090 6, p3-10

[Stephens93] *Controlling Prototyping and Evolutionary Development*
M. Stephens, P. Bates
Proceedings 4th International Workshop on Rapid System Prototyping
28-30 June 1993, Research Triangle Park, USA, p164-185
ISBN 0 8186 4300 5, IEEE Computer Society Press

[Sutcliffe98] *Experience with SCRAM: A Scenario Requirements Analysis Method*
A. G. Sutcliffe, M Ryan
Proc. 3rd International Conference on Requirements Engineering (ICRE98)
Colorado Springs, USA, 1998, p164-171

[Swartout82] *On the Inevitable Intertwining of Specification and Implementation*
W. Swartout, R. Balzer
Communications of the ACM
Vol 25, No 7, p438-440, July 1982

[Swebok03] *Software Engineering Body of Knowledge (SWEBOK)*
http://www.swebok.org/home.html
Accessed 30th July 2003

[Tanimoto87] *Visual Representation in the Game of Adumbration*
Steven L. Tanimoto
1987 IEEE Workshop on Visual Languages
19-21 August, 1987, Linkoping, Sweden, ISBN 91 7870 208 9
IEEE Computer Society Press, p17-28

[Thebaut90] *Marcel: A Requirements Elicitation Tool Utilising Scenarios*
Stephen M. Thebaut, Mark F. Interrante, T. Frederick Burch
Proc. of CAiSE'90: 4th Int. Workshop on Computer-Aided Software Engineering
5-8 December, 1990, Irvine, CA, IEEE Computer Society Press
ISBN 0 8186 2129

[Tickit01] *The TickIT Guide*
British Standards Institution, 2001, ISBN 0 580 36943 9

[Thompson92] *Prototyping: Tools and Techniques - Improving Software and Documentation Quality Through Rapid Prototyping*
M. Thompson, N. Wishbow
SIGDOC-'92 – The 10th Annual International Conference, 1992, p191-9
ACM Press

[TSO01] *The Highway Code*
The Stationery Office Books, 2001
ISBN 0115522905

[Valeric03] *Essentials of Anatomy and Physiology*
C. Valeric C., Tina Sanders
Publisher: F. A. Davis, 2003, ISBN: 0803610076

[VanSchouwen91] *The A-7 Requirements Model: Re-examination for Real Time Systems and an Application to Monitoring Systems*
A. John van Schouewen
Technical Report 90-276
Queen's University, Kingston, Ontario K7L 3N6
January 1991

[Vienneau97] *A Review of Formal Methods*
in Software Requirements Engineering (2nd Ed)
Edited by R.H. Thayer, M. Dorfman
IEEE Computer Society Press, 1997, p324-335

[Visualisation02] Proceedings 13th IEEE Visualization 2002 Conference (VIS 2002)
October 27 - November 01, 2002, Boston, MA, USA,
IEEE Computer Society Press

[Walton94] *Now You See It - Interactive Visualisation of Large Data-Sets*
Jeremy. Walton
Proceedings Share Europe
18 April 1994, Brussels, Belgium, p769-782

[Ward85] *Structured Development for Real-Time Systems*
P. T. Ward, S. J. Mellor
Yourdon Press, 1985

[Webster90] *Scientific Visualisation - Great Hope or Great Hype?*
G. Webster
Proceedings of Computer Graphics '90 Conference
6-8 November 1990, London, UK, p323-330, ISBN 0 86353 253

[Weaver98] *Practical SSADM (Version 4+): A Complete Tutorial Guide*
Philip L. Weaver, Nick Lambrou, Matthew Walkley
Financial Times Pitman, 1998

[Weidenhaupt98] *Scenarios in System Development: Current Practice*
Klaus Weidenhaupt, Klaus Pohl, Matthias Jarke, Peter Haumer
IEEE Software
March 1998, p34-45

[Williams94] *Assessment of Safety Critical Specifications*
Lloyd G. Williams
IEEE Software
January 1994, p51-60

[Zave90] *A Comparison of the Major Approaches to Software Specification and Design*
Pamela Zave
System & Software Specification
Edited by Richard H. Thayer, Merlin Dorfman,
IEEE Computer Society Press, 1990, ISBN 0-8186-8921-8

[Zave97] *Four Dark Corners of Requirements Engineering*
Pamela Zave, M. Jackson
ACM Transactions on Software Engineering and Methodology
Vol 6, No 1, 1997

Appendix A – System Specification Document

Table of Contents

1. Introduction

This document details the overall concepts, sub-systems, software products, interfaces, and capabilities of a system that will enable a user to produce and use prototypes of software systems in the process of requirements validation.

1.1 System Purpose

The purpose of the system is to facilitate the construction, execution and application of prototypes to requirements validation.

The system is founded upon the use of formal specifications. It provides facilities for executing such specifications to produce prototypes. The system then provides additional facilities to visualise the execution of such prototypes with the aim of furthering their effectiveness in the validation process.

1.2 System Scope

It is intended that the system detailed in this document will be applied to the process of acquiring and validating a requirements specification. The scope of the system encompasses the activities necessary to enable a user to:

- Construct specifications of software systems using the formal specification notation Z.
- Transform Z specifications into an executable form that can subsequently be animated.
- Produce two alternative forms of prototype for use in the validation process:
  1. A textually-based prototype, by subjecting the specification to execution.
  2. A visually-based prototype, by applying visual representations to the execution results in real-time. To facilitate this, the system enables a user to compose appropriate visual representations.

2. General System Description

2.1 System Context

To support the production and application of prototypes to requirements validation, the system consists of a process and a set of complementary software tools. In terms of software, the approach comprises three software systems as shown in *Figure A.1*.

The first tool is **TranZit**, which offers full-screen 'WYSIWYG' style editing, syntax analysis, and type checking facilities for Z specifications as well as a transformation system that allows Z specifications to be transformed into an executable form.

The second tool is **ZAL**, a lisp-based environment and notation to provide execution behaviour for Z specifications. The relationship between the two tools is that TranZit is used to capture and format Z specifications, and then subsequently to transform the Z into an executable form, based upon the aforementioned ZAL notation, for processing by the ZAL environment. The ZAL environment possesses an execution engine that interprets ZAL based specification to derive a dynamic and interactive prototype.

The TranZit and ZAL software tools constitute the **RealiZe** approach.

The third tool is **ViZ**, an environment that shall provide support for visualising the execution of ZAL specifications. This software system shall provides facilities to enable a user to compose, edit, and store visual representations then apply them to the results of executing the specification.

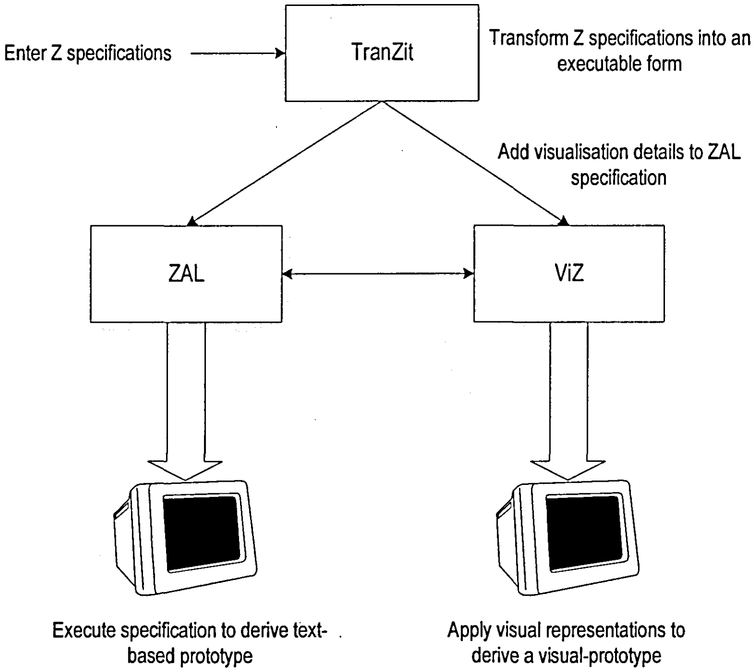

Enter Z specifications
TranZit
Transform Z specifications into an executable form
Add visualisation details to ZAL specification
ZAL
ViZ
Execute specification to derive text-based prototype
Apply visual representations to derive a visual-prototype

Figure A.1. The tools that comprise the system for requirements validation.

2.2 Major System Capabilities

The capabilities of the system can be categorised in terms of the functionality available from the three separate tools.

2.2.1 Capabilities of the TranZit Tool

1. Enable a user to compose and edit Z specification using a WYSIWIG-style editor.
2. Provide syntax analysis and type checking facilities for verifying the specification.
3. Enable the Z specification to be transformed into an executable form based upon the ZAL notation – a notation that supports the execution of a large subset of the Z notation.

2.2.2 Capabilities of the ZAL Tool

1. Provide execution facilities for specifications based upon the ZAL notation to produce prototypes.
2. Enable a user to invoke execution of Z schemas via a simple text-based command-line user interface.
3. Enable a user to interact with the prototype, i.e. enter values for inputs and view the results of execution via a text-based based user interface.

2.2.3 Capabilities of the ViZ Tool

1. Provide support for visualising the execution of ZAL specifications.
2. Enable a user to compose visual representations via a graphical direct-manipulation style user interface.
3. Enable a user to associate visual representations with specification elements to indicate to the software system how to visualise the execution results. This is achieved through modifying the ZAL specification to indicate which specification element to visualise and which visual representation to apply.
4. Enable a user to select schemas and subject them to execution, and view the resulting visualisation of that execution.
5. To provide execution behaviour for the specification, the ViZ system shall interface with the ZAL execution engine, whereby the ZAL specification elements will be processed.

2.3 User Characteristics

It is envisages that there will be two classes of users for the system:

1. The developers of a software system who are concerned with the activity of eliciting, negotiating, and validating requirements. This user class comprises those concerned with developing specifications, composing visualisations, attaching visualisation details to specifications, and executing the specification for the purpose of evaluation.
2. The customers, managers, or other non-development stakeholder of the same software system who are involved with the requirements validation process. These users are concerned with executing the specification, experimenting and interacting with the resulting model, and commenting on its suitability and accuracy.

2.4 Operational Scenarios

The usage of the system is outlined in *Figure A.2*. This shows the capabilities of the system and how the users will exploit these.

3. System Capabilities, Conditions, & Constraints

3.1 The TranZit System

At the most simplistic level, TranZit is a full-screen editor. It offers the same standard editing facilities that can be found in any word-processing software, such as cut, copy, paste, etc. The difference between TranZit and conventional editors is the ability to offer creation and editing of Z specifications. It offers, through the user interface, the full range of Z symbols in accordance with the defined Z standard. To enhance productivity and ease of use, TranZit includes facilities to perform operations such as the automatic generation of Z schema boxes. To this end, TranZit is used to assist in the formalisation process of specification production.

In addition to providing creation and editing facilities, one of the key features of TranZit is its ability to transform a captured Z specification into a representation that facilitates execution.

Central to this is TranZit's Z syntax- and type-checkers and its transformation engine. The syntax-checker is used to verify the correctness of captured Z specifications with respect to the defined Z syntax. In addition, the type-checker is used to verify the correct use of data types throughout Z specifications. Transformation into the executable form cannot proceed until all syntax and type errors are eliminated. Both checking mechanisms are a necessary part of the transformation process, whereby a 'picture' of the structure and syntax of the specification is built up in readiness for transformation. The transformation engine subsequently uses this information together with a database of Z to ZAL rules to produce a syntactically equivalent form.

Transformation results in the production of a specification that is directly executable. This specification is based upon the ZAL notation, which is a lisp-based executable subset of Z. The transformation process is largely automatic. However, there may be instances where a developer may be required to complete the process. This is due to the incompleteness of the ZAL notation with regards to its ability to provide execution behaviour for only a subset (albeit a large subset) of Z. When this situation arises, the developer must 'fill in the gaps' and complete the ZAL specification.

The output of the transformation process is an entity suitable for processing directly by the ZAL execution engine.

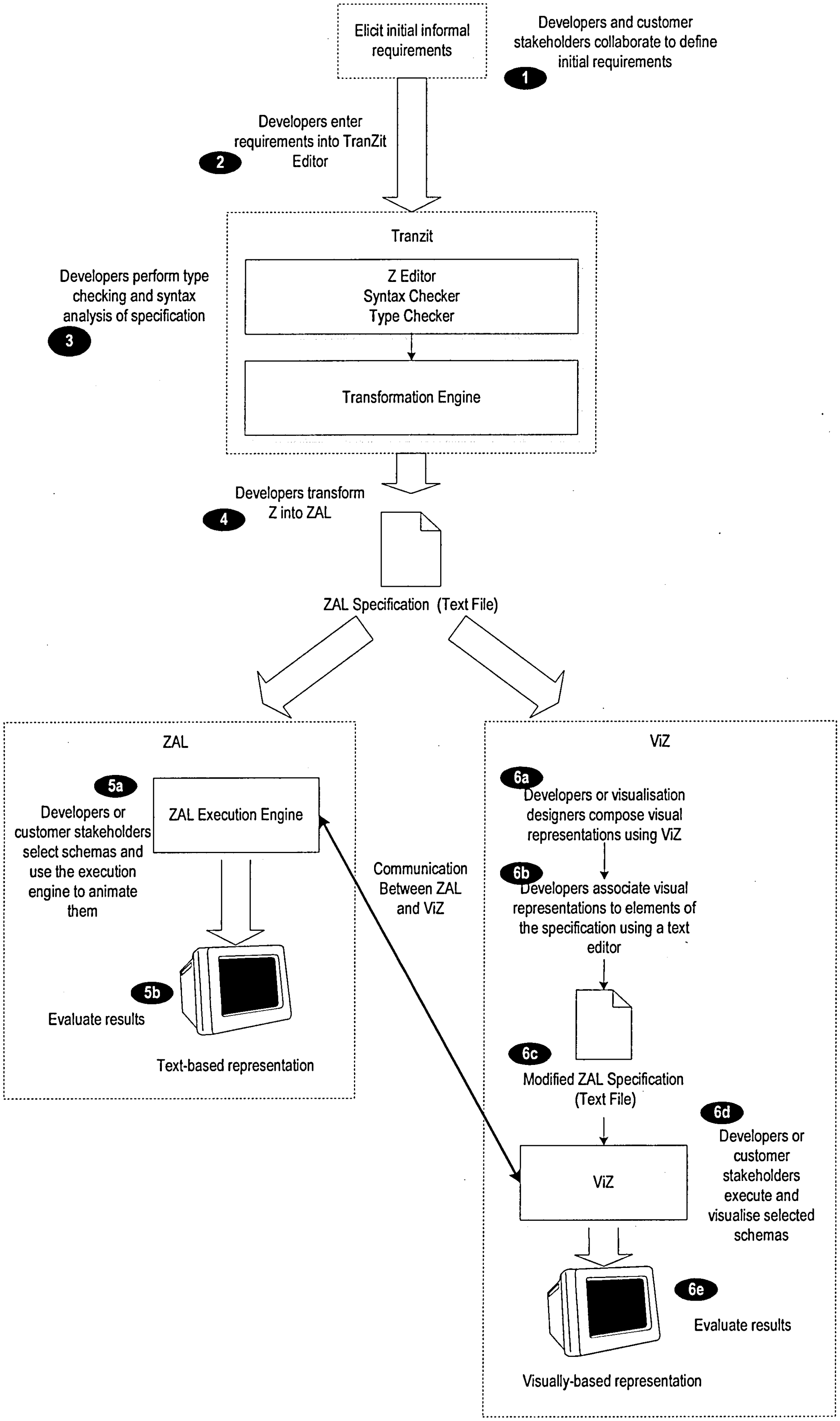

Elicit initial informal requirements
Developers and customer stakeholders collaborate to define initial requirements
1
Developers enter requirements into TranZit Editor
2
Tranzit
Developers perform type checking and syntax analysis of specification
3
Z Editor
Syntax Checker
Type Checker
Transformation Engine
Developers transform Z into ZAL
4
ZAL Specification (Text File)
ZAL
5a
Developers or customer stakeholders select schemas and use the execution engine to animate them
ZAL Execution Engine
5b
Evaluate results
Text-based representation
Communication Between ZAL and ViZ
ViZ
6a
Developers or visualisation designers compose visual representations using ViZ
6b
Developers associate visual representations to elements of the specification using a text editor
6c
Modified ZAL Specification (Text File)
6d
Developers or customer stakeholders execute and visualise selected schemas
ViZ
6e
Evaluate results
Visually-based representation

Figure A.2. The approach to requirements validation.

3.2 The ZAL System

Execution support for prototypes is provided through the "Z Animator in LISP" (ZAL). It is via ZAL that Z specifications can be executed. ZAL comprises a notation and an execution environment.

Execution is performed on an indirect basis whereby Z specifications are transformed (using TranZit) into the ZAL executable form. The syntax and structure of ZAL is deeply rooted in LISP, and a lisp execution system is used to provide execution behaviour. The ZAL notation provides execution behaviour for a large subset of the Z language and includes all of the constructs necessary for writing most practical specifications. The ZAL notation is less abstract than the equivalent Z, but there is essentially an isomorphic correspondence between the two.

The ZAL notation provides for all the data structures in Z, for example, sets sequences, mappings, etc, and provides a wide variety of Z derived operations that may be used on them. ZAL and its underlying execution behaviour reflect the state based nature of Z, in that it allows system state models to be specified and subsequently modified by the operations as execution progresses.

In addition to representing equivalent Z specifications, ZAL also possesses rudimentary input and display capabilities that can be used to set and display the contents of system state variables.

The ZAL environment is subsequently used to process the ZAL specifications. The environment has been developed to include facilities to invoke execution operations and to investigate system states and the contents of data structures and variables.

3.3 The ViZ System

The ViZ system extends the TranZit and ZAL software tools by enabling the execution of ZAL prototypes to be visualised.

The ViZ system is related to the existing software tools and process in such a way as to collect, interpret, and subsequently visualise the results of executing ZAL specifications. The ViZ system consists of:

1. A software system that plays a central role towards supporting of visualisation activities. This system supports the development and production of visualisation details and allows visual representations to be created or imported. In addition, it provides a mechanism for the long-term storage of visual representations. It also promotes the concept of correspondence, through an assurance mechanism, between the meaning of visual representations and underlying specifications.
2. A means to enable associate visualisations to specifications to indicate which elements of that specification are to be visualised and which visual representation to apply. This may be described in terms of an interface between the required visualisation details, the specification, and the visualisation renderer.
3. User support in terms of a process that describes the activities and steps required to develop and use visual prototypes in validating requirements specifications.
4. Separate software packages to enable users to compose and edit high-quality visual representations (i.e. graphics packages), and allow them to edit specifications to add visualisation details (a text editor).

Figure A.3 presents the relationships between the software components.

4. System Interfaces

4.1 Interface Between Visual Representations and Specifications

To enable appropriate visualisations of the execution of ZAL specifications to be rendered, the software-based renderer must possess enough information as to which specification element to visualise and which visual representation to apply.

To this end, it is necessary to enable such details to be specified by attaching them to the specification itself. In this approach, the specification would then contain everything the software required in order to visualise its execution.

It is unnecessary and infeasible and to include all visualisation details in the specification. Therefore the details must be applied in the specification in a minimal form, which also means that much the visualisation details required must be stored outside of the specification.

To enable this, an interface between visualisation details and the specification must be established. The interface should possess the following characteristics:

1. 'Independence' - the mechanism for connecting visualisation details to a specification should ideally be minimal, i.e. the mechanism should require that the least amount of visualisation detail be implanted into a specification.
2. The storage mechanism should allow the majority of visualisation information to be separated from the specification.
3. Syntactic enhancements to the ZAL notation should enable such visualisation details to be stored separately to be 'referenced' from a specification.

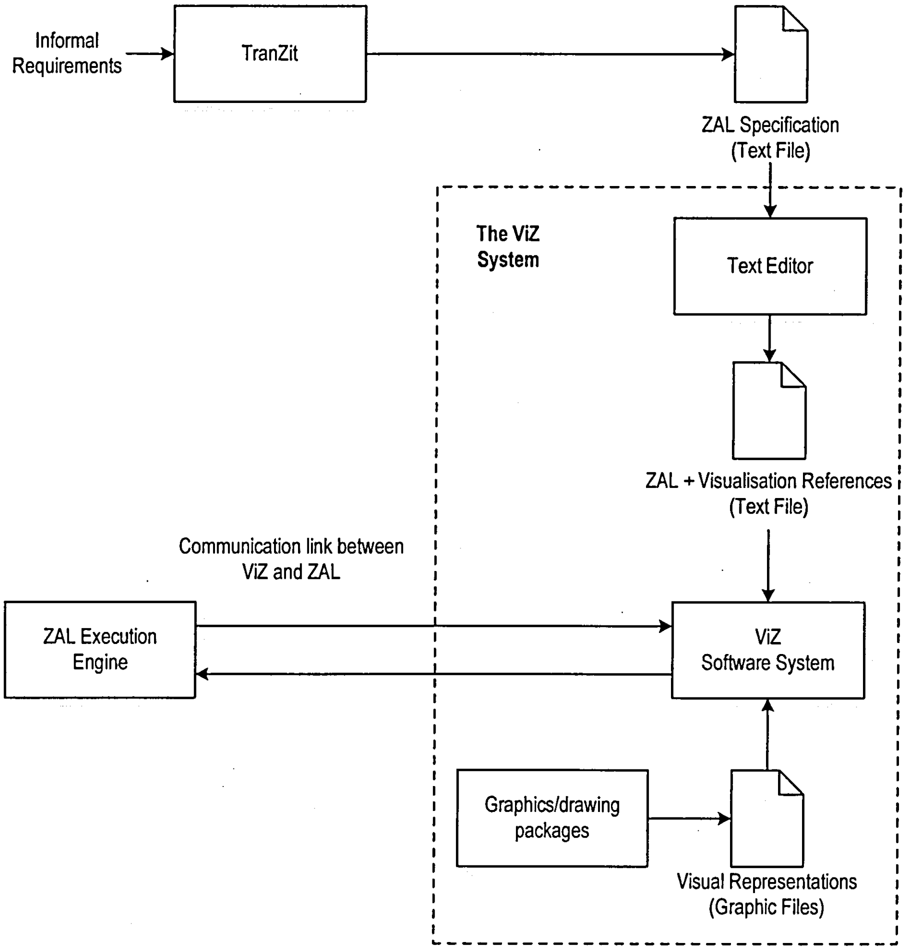

Informal Requirements
TranZit
ZAL Specification (Text File)
The ViZ System
Text Editor
ZAL + Visualisation References (Text File)
Communication link between ViZ and ZAL
ZAL Execution Engine
ViZ Software System
Graphics/drawing packages
Visual Representations (Graphic Files)

Figure A.3. The relationship between the software components and interfaces in the ViZ system.

In addition, visualisation details and syntactic enhancements shall be amenable to processing by the ViZ software system in order for it to render appropriate visual representations.

The relationship between the elements of the interface and the ViZ system components is shown in *Figure A.4*.

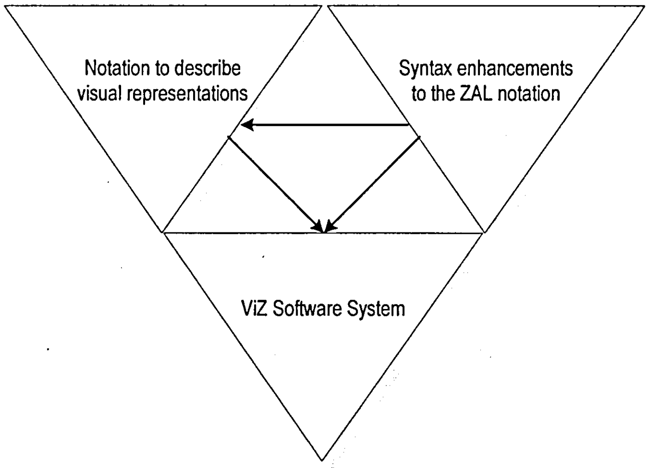

Notation to describe visual representations
Syntax enhancements to the ZAL notation
ViZ Software System

Figure A.4. *The relationship between the 'interface' between specifications and visual representations.*

Appendix B – ViZ Software Requirements Specification Document

Table of Contents

1. Introduction

1.1 Purpose

The purpose of this document is to present in a precise manner, all software requirements deemed necessary in a system for visualising the execution of ZAL specifications. This specification will describe the functional, non-functional, and interface requirements of a software system that is intended to support the process of visualising the execution of ZAL specifications.

This specification is intended to be the baseline to supply sufficient and appropriate information with which a suitable design and subsequent implementation can be derived.

1.2 Scope

The objective of this SRS is to describe the software requirements of the ViZ system. The system will provide the support for composing and editing visual representations, enable such visual representations to be attached to ZAL specifications, then process the resulting specifications with a view to executing them and visualise the results of this execution. The software will also provide support for storing visual representations in a repository.

1.3 Definitions

DDE	- Direct Data Exhange
MS	- MicroSoft™
ZAL	- Z Animation in Lisp

1.4 Document Overview

This document has three major sections. Section 1 (Introduction) provides an overview of this SRS document.

Section 2 describes the product that will be produced. This includes:

- Product perspective
- Product activity
- User characteristics
- Constraints
- Assumptions
- Requirements subsets

Section 3 addresses the specific requirements of the ViZ system. This includes:

- Functional requirements - these include inputs, process aspects, and outputs for each primitive process inherent in the system.
- External interface requirements

2. General Characteristics

This section introduces the characteristics of the ViZ software system. It also describes its relationship to other software systems.

2.1 Product Perspective

The ViZ software system is a part of set of software tools and processes that facilitate the development of prototypes for requirements validation purposes. The relationship between the ViZ software system and the other software systems are described in greater detail in the *System Specification Document* in *Appendix A*. The relationships are summarised in *Figure B.1.* The figure describes the abstract architecture of the system - components in broken lines indicate generic software tools, i.e. any appropriate non-specific software tools that may be used to fulfil the task as indicated.

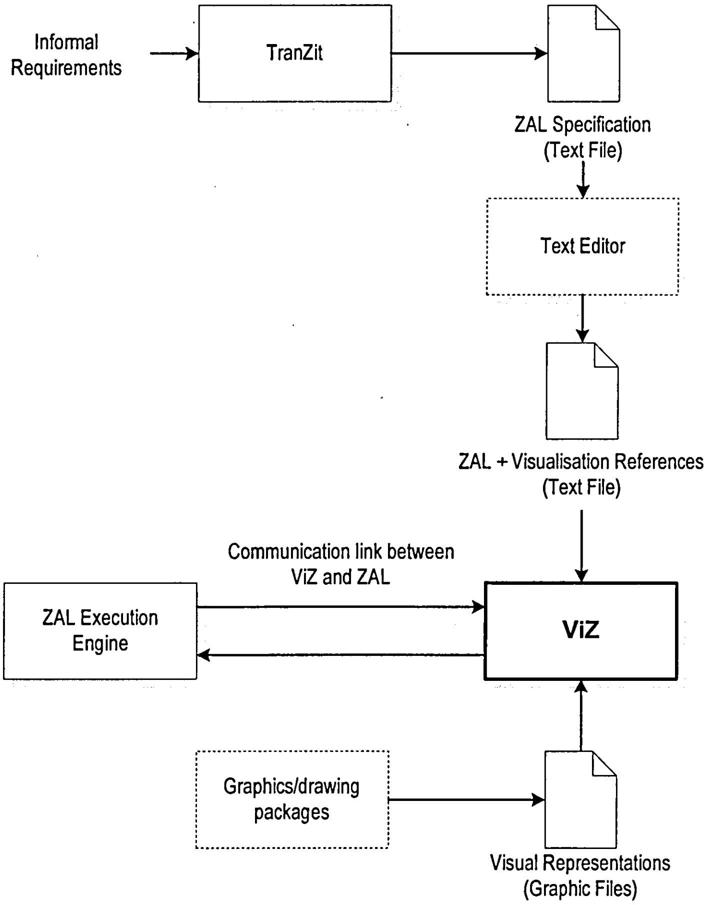

Informal Requirements
TranZit
ZAL Specification (Text File)
Text Editor
ZAL + Visualisation References (Text File)
Communication link between ViZ and ZAL
ZAL Execution Engine
ViZ
Graphics/drawing packages
Visual Representations (Graphic Files)

Figure B.1. The relationships between ViZ and other relevant software packages in the system.

The requirements described in the remainder of this document relate only to the ViZ software system.

2.2 Interfaces

2.2.1 Software Interfaces

The ViZ software system is part of a larger overall approach that facilitates the production of prototypes for requirements validation purposes. To this end, ViZ interfaces with the other software systems that form part of the approach. These, shown in *Figure B.1*, are:

TranZit - This tool is designed to enable a user to enter and edit Z specifications directly. In addition, it translates Z specifications into ones based upon the ZAL notation which facilitate execution. The output from TranZit is a text file containing a ZAL specification.

Text-editor - The "System Specification Document" describes an interface between ZAL specifications and visualisations in the form of an extended specification syntax. To modify a standard ZAL specification in accordance with the syntax, in order to associate visual representations with specification elements, the user may employ any generic text-editor. The interface between the ViZ system and the text editor should be based upon a file-sharing mechanism that enables ViZ to import modified ZAL specifications.

ZAL - This component performs the evaluation/execution of ZAL specifications. The interface between ZAL and ViZ shall be based upon a software-based communication link and protocol.

Graphics Packages - ViZ should enable visual representations to be associated with specifications and then subsequently render visual representation during execution. ViZ should provide rudimentary facilities for composing visual representation, but when more advanced representations are required (for example photo-realistic quality images), then other graphics packages can be used to generate them. To this end, any generic graphics-package may be used. The interface between ViZ and the graphics packages should be based upon a file-sharing technique, whereby files created by the packages should be imported and stored as visual representations.

It is envisaged that these external software systems be resident in memory at the same time as ViZ and execute in parallel to the ViZ system. It is not necessary for distributed execution mechanisms to be defined and implemented at this time.

2.2.2 Software Communication Interface

A software-based communication link between ViZ and the ZAL execution engine is required to enable ZAL to process and execute the ZAL components of a specification and return the results back to ViZ for visualising. To this end, the communications mechanism and protocol to support this should be based upon MS Windows Direct Data Exchange (DDE) interface. The DDE interface is a shared memory communication technique that enables two software packages to send and receive data.

2.3 User Characteristics

The users of the ViZ system consist of:

1. The developers of a software system who are concerned with the activity of eliciting, negotiating, and validating its requirements. This user group concerns though involved with developing specifications, composing visualisations, attaching visualisation details to specifications, and executing the specification for the purpose of evaluation.

2. The customers, managers, or other non-development stakeholder of the same software system who are involved with the requirements validation process. This group involves those concerned with executing the specification, experimenting and interacting with the resulting model, and commenting on its suitability and accuracy.

2.4 Product Functions

This section lists the functions that should be available in the ViZ system.

➢ The product shall provide support for the execution of ZAL specifications and facilitate their subsequent visualisation.

➢ Through an appropriate user interface, the product should:

- enable a user to select a specification for execution/visualisation.
- enable a user to invoke execution of a specification and initialise the ViZ system in readiness for execution and visualisation.
- enable a user to select a single ZAL schema from a list of schemas that are defined in a specification. The product shall then execute this selected schema.
- provide a means of entering input values to schemas when necessary.
- enable a user to pause and resume the execution, and terminate it when desired.
- indicate the progress of executing the selected schema.

➢ The product should display the resulting visualisations and the results of execution on the screen.

➢ The product should provide facilities to enable a user to compose and edit visualisations, including appearance and motion components.

- The product should enable a user to compose appearance components from basic geometric shapes or more advanced visual representations based on photo-realistic quality bitmap images by importing such images as files from external graphics packages.

- The product should enable visualisation components to be stored in the long term for use in current and future projects.

- The product should enable a user to classify visualisation components in respect to the application/project to which they will be used, thus promoting the concept of correspondence.

- The product should pre-process the modified ZAL specification to initialise and prepare the ViZ system in readiness for subsequent schema execution activities.

- The product must execute selected schemas.

- The product must render visualisations of the desired execution elements.

- The product should evaluate ZAL expressions by passing them to the ZAL execution engine.

- The product should establish communications between itself and the ZAL execution engine in order to evaluate ZAL specifications.

Figure B.2 shows the intended dependencies and relationships between the various components and the user's involvement. The figure also shows each component's requirement number in accordance to the above list.

2.5 Assumptions & Dependencies

The product is dependent upon the MS Windows operating environment being present on the target machine.

3. Specific Requirements

The specific requirements for the ViZ product are described below.

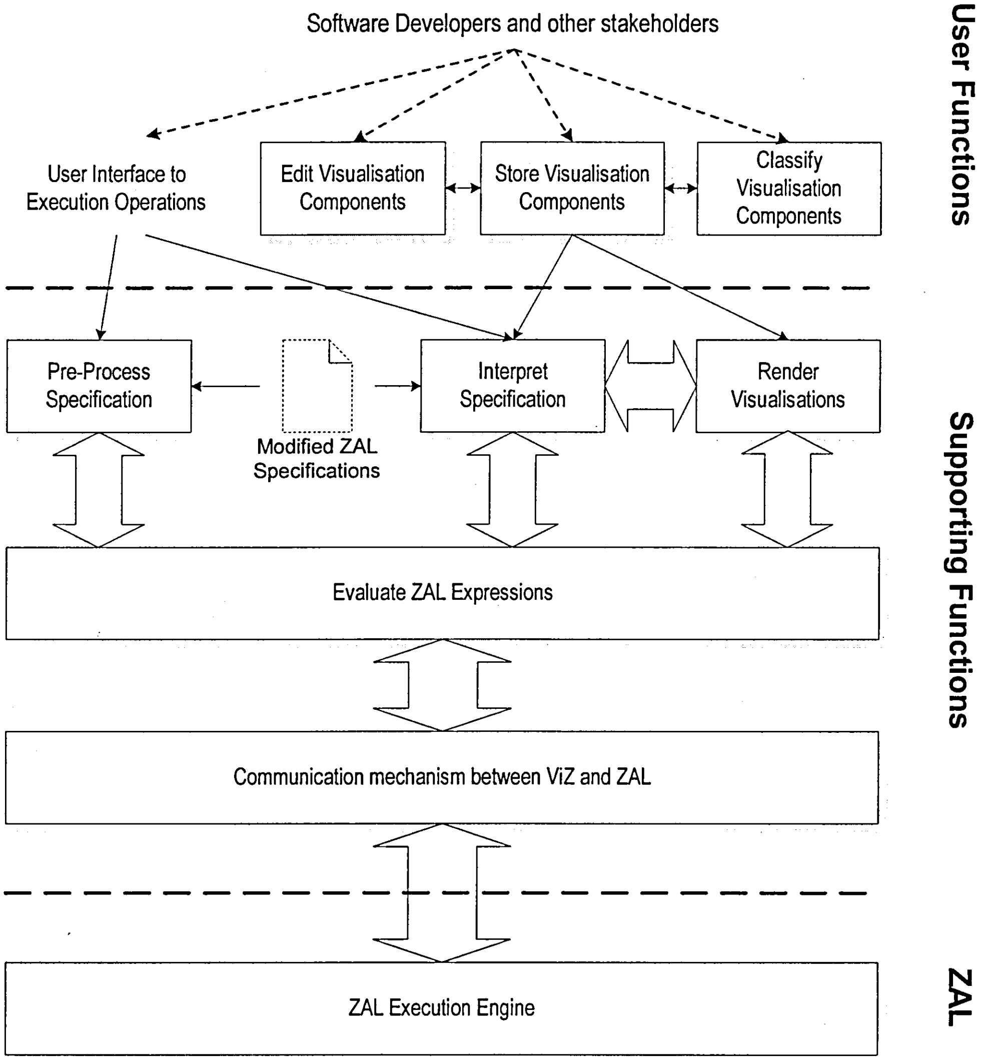

Software Developers and other stakeholders
User Interface to Execution Operations
Edit Visualisation Components
Store Visualisation Components
Classify Visualisation Components
User Functions
Pre-Process Specification
Modified ZAL Specifications
Interpret Specification
Render Visualisations
Evaluate ZAL Expressions
Communication mechanism between ViZ and ZAL
Supporting Functions
ZAL Execution Engine
ZAL

Figure B.2. Relationships and logical dependencies between the components of the product.

3.1 Pre-Processing the Specification

The product will pre-process a ViZ specification, once, to initialise and prepare the ViZ system in readiness for subsequent schema execution activities.

The software should parse the entire specification to determine syntax validity and to initialise any global data definitions.

Inputs

- ViZ specifications as a text file.

Processing

- To pre-process a specification, ViZ should scan the specification and declare and initialise global system-state variables that are encountered by passing them to the ZAL execution engine (through the ZAL expression processor), whereby the system-state shall be maintained.
- The software will also build a corresponding symbol table that contains the names of the system-state variables and their associated types, and the names of the schemas in the specification. The symbol table should also record the locations of each schema within the specification file.

- A syntax check on the visualisation references given in the specification should be performed, i.e. determine the existence within them system of any visualisation-unit referenced within the specification, i.e. determine if the visualisation-unit has already been defined and stored. Syntax checking of ZAL operations should be deferred until processing of expressions, and shall be performed by the ZAL execution engine.
- ViZ should perform a check on the usage of visualisation references in accordance to their applicability to the specification. This check is required in facilitate correspondence between the meaning of a specification and the meaning of associated visualisations.

Outputs

- A symbol table, containing:
  i) Names of the system-state variables.
  ii) The types associated with each variable.
  iii) The names of the schemas in the specification.
  iv) The position within the specification file, of each schema.
- Any error messages generated from the syntax validation check or the correspondence check.

3.2 Enable the User to Select a Schema

After pre-processing the specification, the ViZ system shall enable a user to select a single schema from the whole specification by presenting a choice of schemas. To this end, a suitable user interface will be required. The user shall be given the opportunity to completely exit the execution facilities at this point.

Inputs

- The user's selection (when the menu has been presented)
- The symbol table (to obtain the names of the schemas in the given specification).

Processing

- The choice of schemas should be presented in the form of a list of names (obtained from the symbol table constructed in the parsing activities) that the user might select.

Outputs

- The name of the selected schema.

3.3 Interpret The Selected Schema

The ViZ toolset must interpret a selected schema. Interpretation activities form a major function of the ViZ toolset and are concerned with processing specifications and orchestrating the activities towards their visualisation.

The toolset should interpret specifications to separate the ZAL elements from the visualisation details. It should then pass the ZAL components to ZAL for evaluation, and render any results that are returned.

- The toolset should indicate the position in the specification where execution is currently occurring. It should present a section of the current schema along with a visual indicator.
- A user should be able to pause the execution (and subsequently resume) or terminate it. Suitable user-interface features should be provided to facilitate this.

Inputs

- ViZ specification as a text file.
- The name of the selected schema to execute.
- The symbol-table.

Processing

To interpret a specification, the product shall

- Locate the desired schema in the specification file (from the position recorded in the symbol table created by parsing the specification).
- Execute any schema inclusions. This activity shall recursively execute the schemas encountered in the schema.
- Enable a user to enter input data into a schema, when requested by the specification.
- The schema will then be processed by evaluating each expression deterministically (i.e. line-by-line, in the sequence specified).

When processing each expression the following activities shall take place:

- Determine if the expression is to be visualised, i.e. check for the existence of any scenes referenced by the expression.
- Check for expressions that are used to 'cement' expressions together – i.e. 'or' or 'and' expressions. If these are found, they shall not be visualised but instead use them as the basis to calculate the results of executing the expressions.
- If a reference to a scene is not found, then evaluate the entire expression. Move onto the next expression.
- If a scene is found, visualise the expression.
- The specification processor shall record the results of executing each expression. All expressions are treated as predicates by the ZAL execution engine and their evaluation returns either 'true' or false', the product shall record these values returned from each expression. These values shall then be used in the calculation of expression conjunction or disjunction operations ('and' and 'or') that are used to cement expressions together. The results of evaluating all the operations in the schema shall then be returned as an output from the execution process.
- If any errors are encountered (as indicated by the ZAL expression evaluator) then terminate the execution and display a corresponding error message.
- This process should continue until all expressions in the schema are executed, an error occurs or the user terminates execution. When complete, the user shall be given the opportunity to select another schema for execution, or terminate execution activities.

Outputs

- The display of the on-going progress of the execution.
- The predicated result of executing the schema – a 'true' or false' value depending upon the return values calculated from the results of evaluating each expression and the way in which they are combined.
- If an error is detected during execution then inform the user of the error by presenting it on the display.

3.4 Facilitate the Rendering of Visualisations

The ViZ system must facilitate the rendering of visualisations. To do this, it should apply the details contained in a given scene, as attached to an expression.

The product should provide visualisation for two types of expressions: 1) ZAL expressions; 2) specific output-oriented expressions that are used to display the contents of system-state variables only.

Rendering of the appearances should be based upon the applying the details specified in the appearance component to the values of the system-state variables returned from the ZAL execution engine. This is to facilitate execution-driven visualisation.

Inputs

- A complete expression, including reference to the required scene.

Processing

- To provide visualisation for ZAL expressions, the system should:
  i) Obtain the value of each system-state variable in the expression from the ZAL execution engine (this is required since the system-state is maintained by the execution engine itself and not ViZ).
  ii) Perform type checking on the types of the system-state variables in the ZAL operation and the types of appearance-motion pairs attributed to them in the given scene.
  iii) Visualise these elements by applying the appearance-motion pairs that are specified in the 'before' section of the given scene. Apply the appearance-motion pairs on a first-come-first-served basis.
  iv) Evaluate the ZAL part of the expression. If errors are detected, then terminate the visualisation process, and return any error conditions.
  v) Perform type checking on the types of the value returned from the evaluation of the expression and the types of appearance-motion pairs attributed to them in the given scene. If the type checking is unsuccessful then return an error and terminate execution, otherwise continue.

  vi) Perform visualisation of the results of executing the operation. The renderer shall apply the first appearance-motion pair in the 'after-state' section of the visualisation-unit to the result.
  vii) If the expression contains any additional variables for visualisation, then apply the rest of the appearance-motion pairs to them on a first-come-first-served basis. This activity will entail obtaining the values of these variables from the ZAL execution engine. Perform type-checking during this activity.
- When rendering appearance-motion pairs, the appearances should be rendered at the point specified by the location in the first node in the motion sequence (as specified in the motion component of an appearance-motion pair). If subsequent nodes exist in the motion sequence then the visualisation renderer shall animate the appearance along the path described by the nodes.
- If more than one appearance exists (i.e. more that one system-state variable being visualised), then animate the appearances simultaneously.

Outputs

- Rendered visualisations.

3.5 Facilitating Storage of Visualisation Components

Though a repository, the ViZ toolset should enable all visualisation components to be stored in the long term to enable them to be used in current or future projects. The repository should also promote correspondence, through an assurance mechanism, between the meaning of a visual representation and the meaning of an underlying specification.

Inputs

- User inputs to select/create/delete/edit visualisation components and applications.

Processing

- The ViZ toolset shall enable the long-term storage of individual appearances, individual motions and complete expression level scenes (i.e. sets of appearance-motion pairs as applied to whole expressions).
- The product will also be instrumental in satisfying the requirement to provide correspondence between the meaning of a specification and the meaning of an associated visualisation. To this end it shall enable containers known as "Applications" to be defined. These should enable a user to specify which visualisation components can be used with which specifications.
- The toolset should enable a user to edit visualisation components. To this end, a list of visualisation components shall be provided to the user, then allow a user to select and subsequently edit the desired component.

- The toolset should enable a user to delete and duplicate application containers and visualisation components.
- The toolset should enable type information to be associated with a given appearance component.

Outputs

- A user interface to depict the arrangement and structure of the repository and its contents.

3.6 Enabling Visualisation Components to be Composed and Edited

The product shall enable a user to compose and edit the three aspects of a visual representation, namely:

- Appearances
- Motions/dynamic components
- Scenes/expression level visualisations

The editors must make use of appropriate user interface techniques to promote usability, including the use of graphical direct-manipulation interfaces. The visualisation composition and editing functions are described below.

3.6.1 Enabling Appearances to be Composed and Edited

The product should enable the outward face of a visualisation to be defined from a combination of basic geometrical shapes, bitmaps, and text elements.

Inputs

- User inputs to create/edit appearance components.
- The name of a previously created appearance (if editing of an existing appearance is taking place).

Processing

- The appearance editor should present an image(s) of the appearance as it is being developed via a graphical user interface.
- The product should enable a user to attribute a name to the appearance for identification purposes.
- Completed appearances should be stored long term.
- The product should allow previously defined appearances to be retrieved from storage and edited.

Outputs

- User interface for editing appearance components.
- Completed appearance components to store in long-term storage.

3.6.2 Enabling Motions to be Composed and Edited

The ViZ toolset should enable a user to define the on-screen position and motion of an appearance.

Inputs

- User inputs to create/edit motion components.
- The name of a previously created motion (if editing of a previously created motion is taking place).

Processing

- To allow the movements of an appearance to be described, the motion editor should allow such movements to be described in terms of a series of nodes. The paths between the nodes describe the path of the appearance when then visualisation is rendered.
- The product should facilitate the editing of previously defined components motion components.

- The product shall provide an interactive direct-manipulation style interface with which to place and move nodes.
- The product should enable a user to attribute a name to the appearance for identification purposes.
- The visualisation editor should enable a user to specify the speed of an appearance as it traverses the nodes.
- Completed motions should be stored long term.
- The product should also allow previously defined motions to be retrieved from the component repository and edited.

Outputs

- User interface for editing appearance components.
- Completed motion components to store in long-term storage.

3.6.3 Enabling Visualisation Scenes to be Composed and Edited

The ViZ toolset should enable scenes, i.e. complete expression-level visualisations, to be defined and edited, via manipulating the appearance and motion components. This should be performed using an interactive graphical user interface.

Inputs

- User inputs to create/edit scenes.
- The name of a previously created scene (if editing of a previously created scene is taking place).

Processing

- The ViZ toolset should employ a suitable interactive user interface to enable the structure and content of a scenes to be displayed while undergoing editing.
- The product should enable a user to attribute a name to the scene for identification purposes.
- Completed scenes should be stored in a long-term storage repository.
- The visualisation editor should also allow previously defined visualisation-units to be retrieved from long-term storage and edited.
- This software component should also enable a user to classify/partition scenes in accordance with the software development project to which they will be applied – this forms part of the assurance mechanism.

Outputs

- User interface for editing scenes.
- Completed scenes stored in the component repository in readiness for subsequent application.

3.7 Evaluate ZAL Expressions

The ViZ software tool will include a component, known as the ZAL expression processor, to be responsible for handing expressions to ZAL for evaluation. This component interfaces with the communication subsystem.

Inputs

- An expression string from the visualisation engine.

Processing

- The expression processor passes a string containing a ZAL expression to the ZAL execution engine for processing.
- The processor must wait for any results that are returned from the ZAL execution engine.

- If the ZAL execution system reports an error, then the expression evaluator should signal that an error has occurred to the rest of the ViZ system, and terminate execution and visualisation activities.
- To pass the expression to the ZAL execution engine, the expression processor should establish a software-based communications channel between itself and the execution engine (refer to Section 3.8).

Outputs

- The results of executing the given expression.
- Any errors from the execution.

3.8 Establish Communication between ViZ and ZAL

To pass a ZAL expression to the ZAL execution engine and retrieve the results, a software-based communications channel between ViZ and the ZAL execution engine must be established.

To facilitate the communication between the two software systems, the product should use the MS Windows DDE interface and protocol that is designed to enable two memory-resident applications to share data.

Inputs

- An expression string to pass from the ZAL expression processor to the ZAL execution engine.
- A result string to pass from ZAL execution engine to the ZAL expression processor.

Processing

- The product should appropriately package the string to be passed to ZAL according to the protocol defined by the DDE specification.
- The product should invoke the required MS Windows OS routines to initiate the transfer.
- The product must wait for the ZAL execution engine to return the results of executing the given expression.

Outputs

- The result string - passed back to the ZAL expression processor.

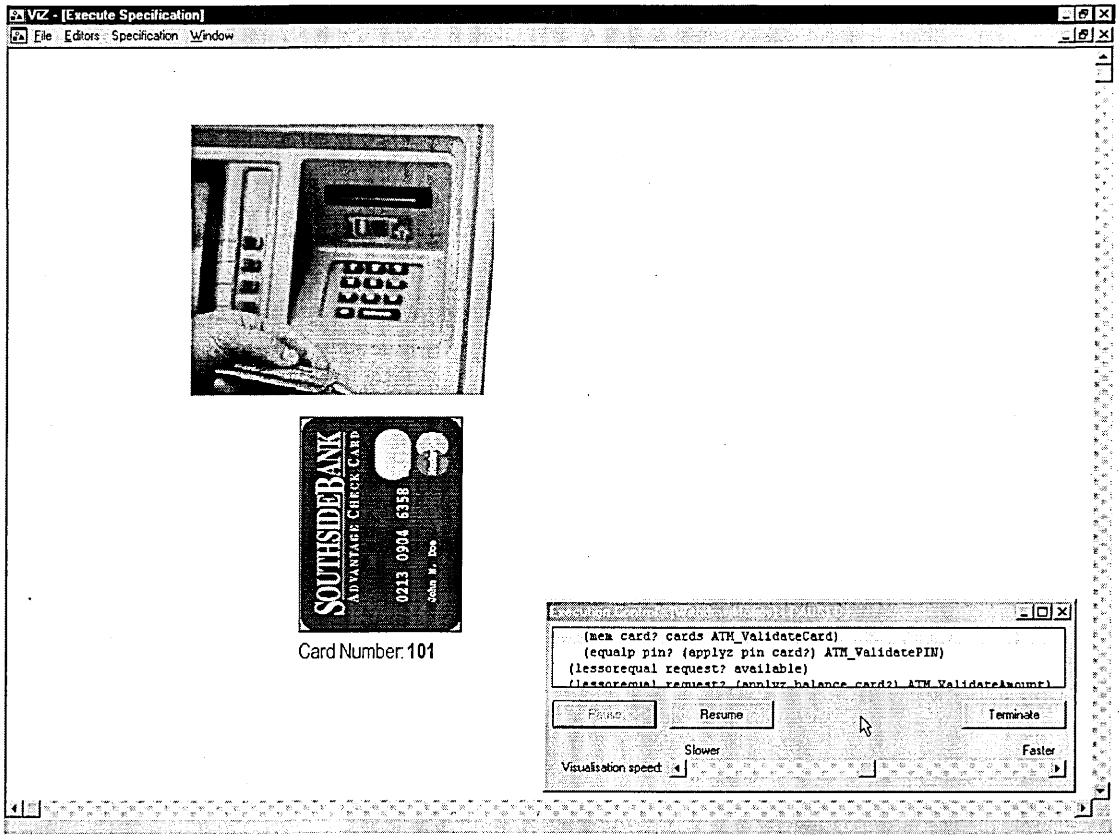
Appendix C – Results of ATM Visual Prototype Execution

This Appendix presents the complete sets of results of executing the ATM visual prototype from *Section 4.1*. The set contains the results of executing the prototype in the context of successful money withdrawal.

This scenario embodies the activities involved in successful dispensing an amount of money. The activities involved, and the associated visual scenes, are shown in *Table C.1*.

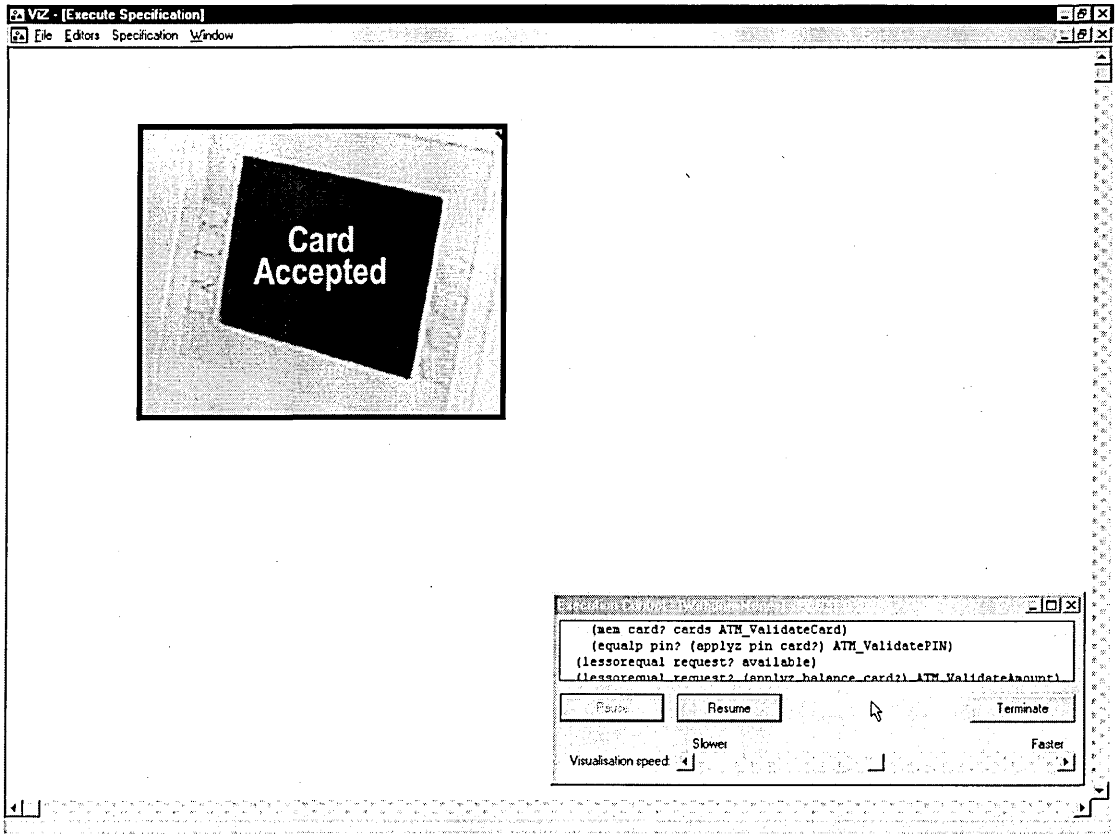
Scenario Activity	Possible Scenes
1. Customer inserts card into the card reader	Scene 1. Show customer inserting card into machine. (used to show the state of the system before executing the expression).
2. Card is validated	Scene 2. Show results of card validation (used to portray the state of the system after the expression is executed).
3. Customer enters PIN	Scene 3. Show customer entering PIN on keypad. (used to show the state of the system before executing the expression).
4. PIN is validated	Scene 4. Show results of PIN validation. (used to portray the state of the system after the expression is executed).
5. Customer enters required amount of money	Scene 5. Show customer entering requested amount using keypad. (used to show the state of the system before executing the expression).
6. Validate amount	Scene 6. Show results of amount validation. (used to portray the state of the system after the expression is executed).
7. Eject card	Scene 7. Show card being ejected
8. Dispense money	Scene 8. Show money being dispensed. (used to portray the state of the system after the expression is executed).

Table C.1. *The activities and scenes in the successful money withdrawal scenario.*



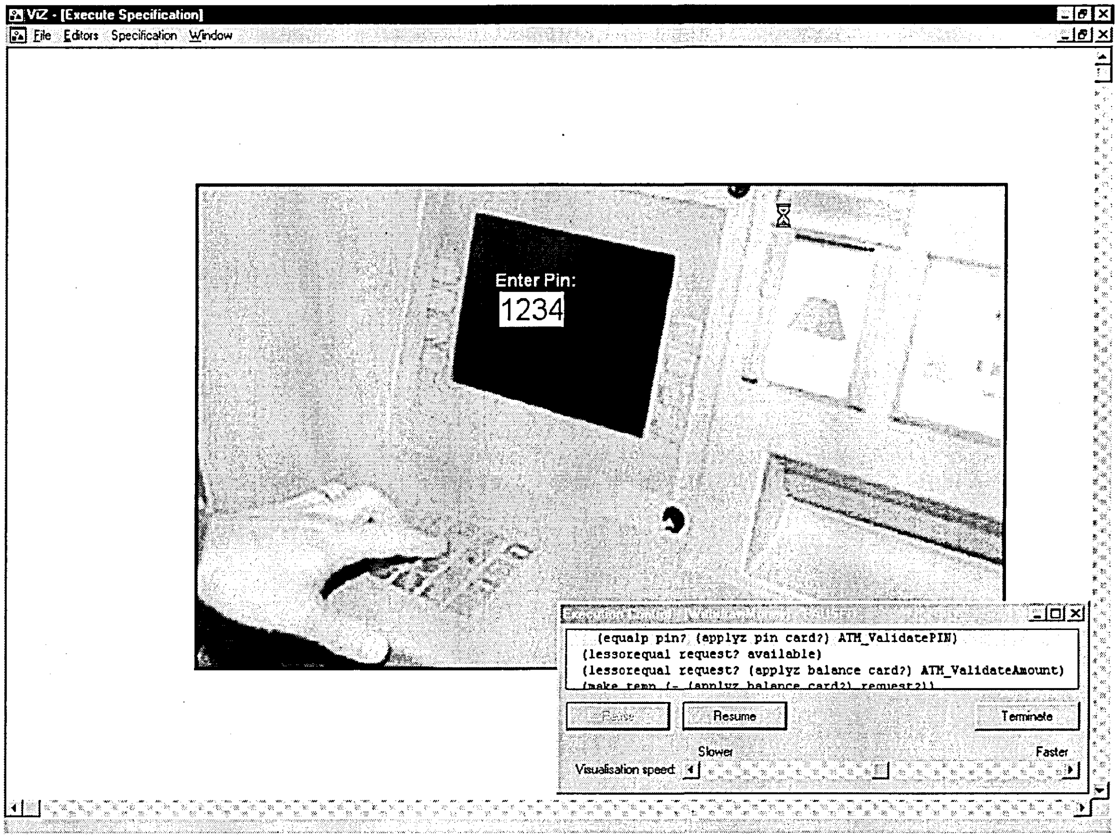
```
    (mem card? cards ATM_ValidateCard)
    (equalp pin? (applyz pin card?) ATM_ValidatePIN)
  (lessorequal request? available)
  (lessorequal request? (applyz balance card?) ATM_ValidateAmount)
```

Scene 1. Insertion of the card – the card number is '101'.



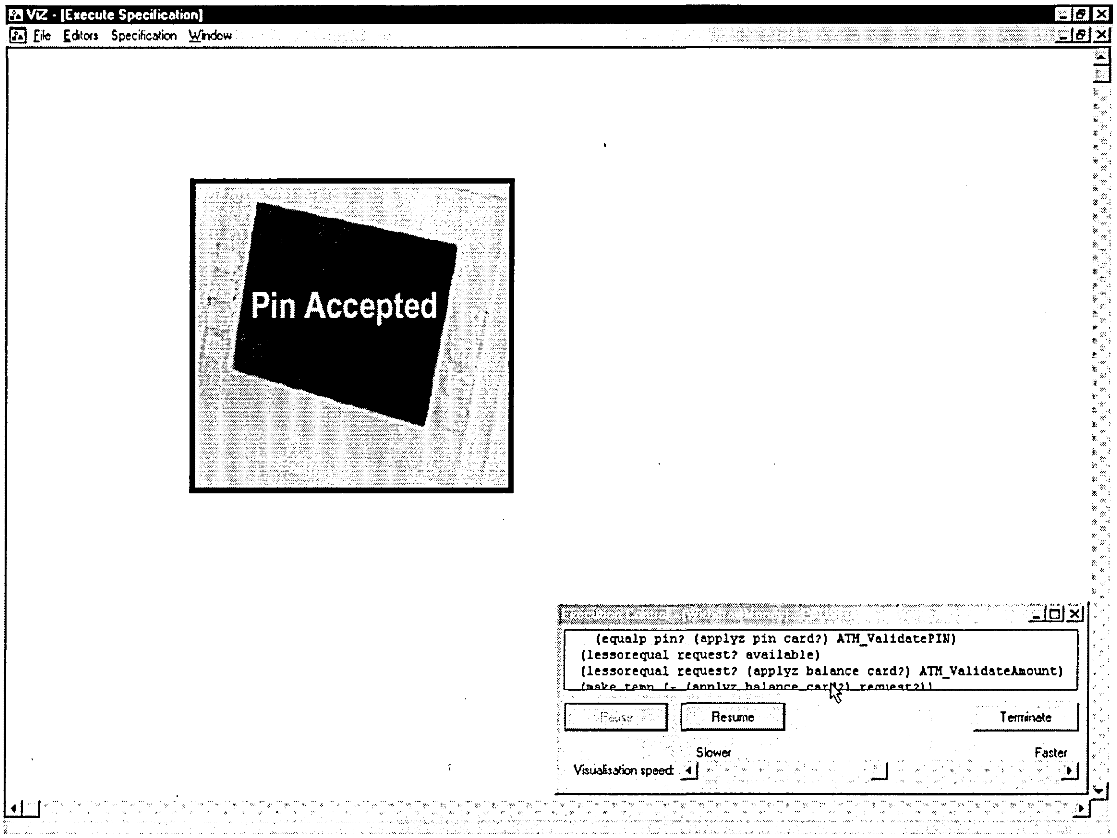
```
    (mem card? cards ATM_ValidateCard)
    (equalp pin? (applyz pin card?) ATM_ValidatePIN)
  (lessorequal request? available)
  (lessorequal request? (applyz balance card?) ATM_ValidateAmount)
```

Scene 2. Card accepted



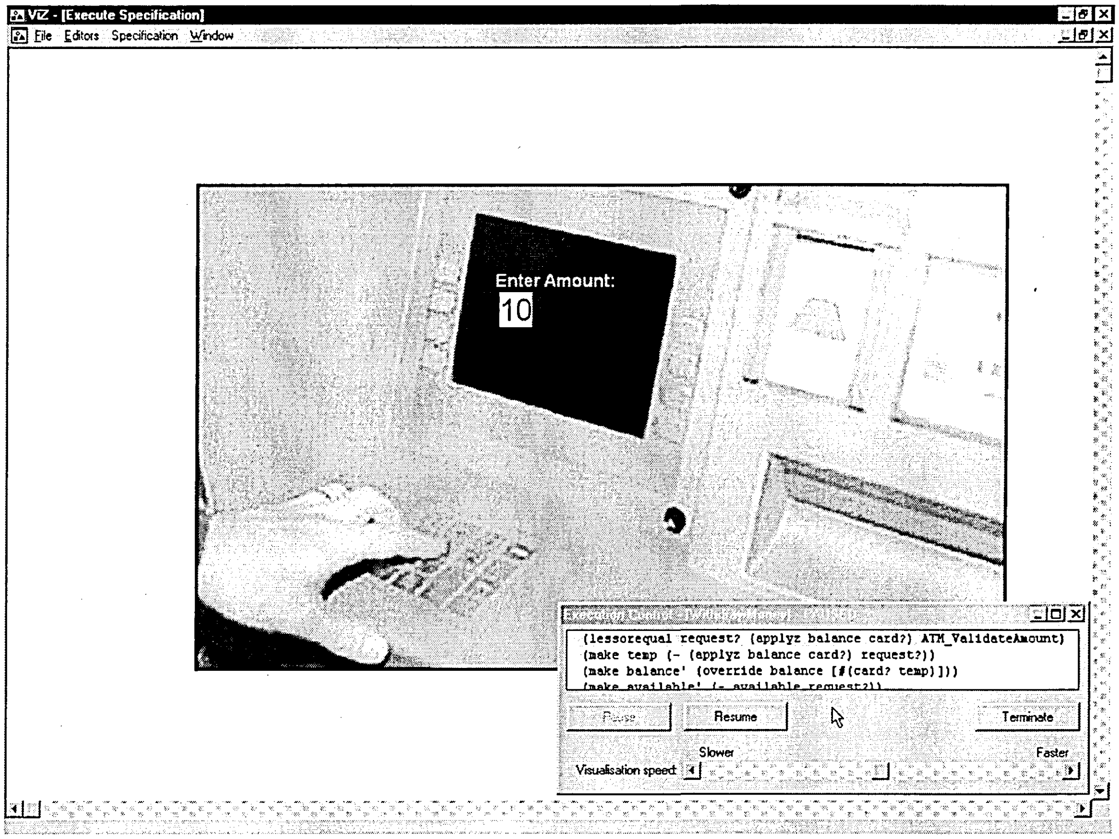
```
  (equalp pin? (applyz pin card?) ATM_ValidatePIN)
(lessorequal request? available)
(lessorequal request? (applyz balance card?) ATM_ValidateAmount)
(make temp (- (applyz balance card?) request?))
```

Scene 3. Entering the Pin.



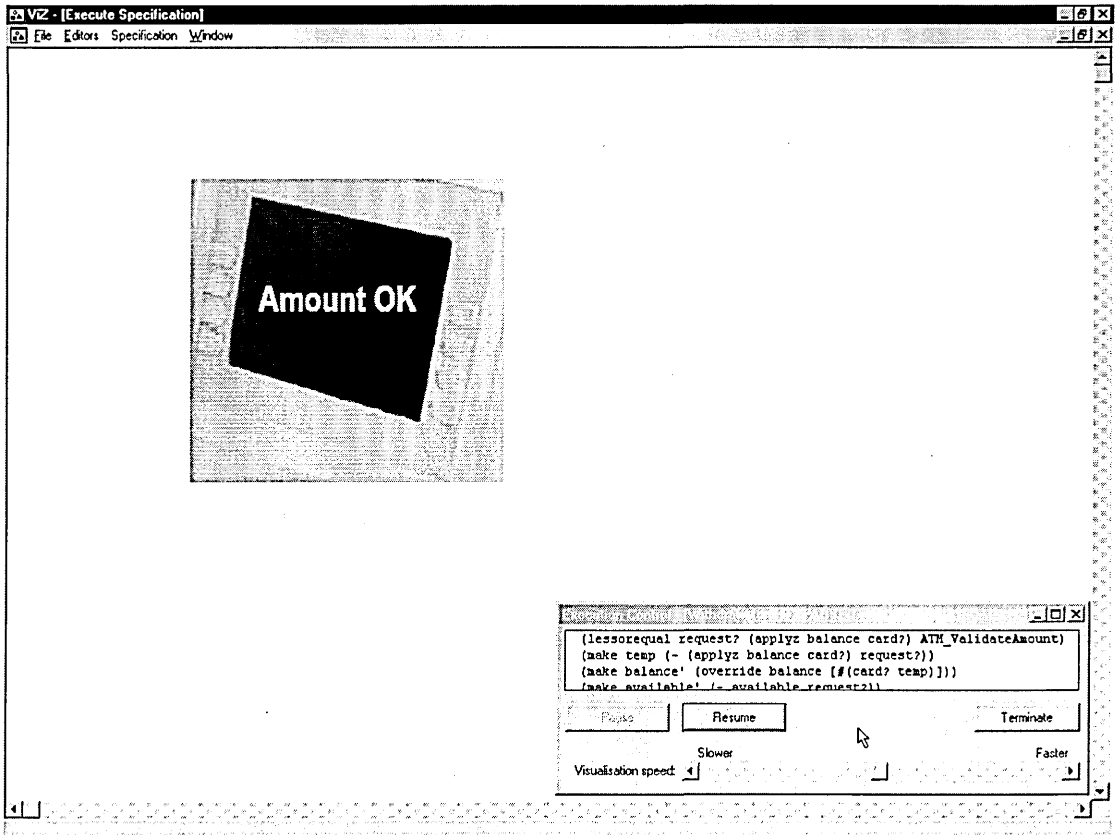
```
  (equalp pin? (applyz pin card?) ATM_ValidatePIN)
(lessorequal request? available)
(lessorequal request? (applyz balance card?) ATM_ValidateAmount)
(make temp (- (applyz balance card?) request?))
```

Scene 4. The Pin is accepted.



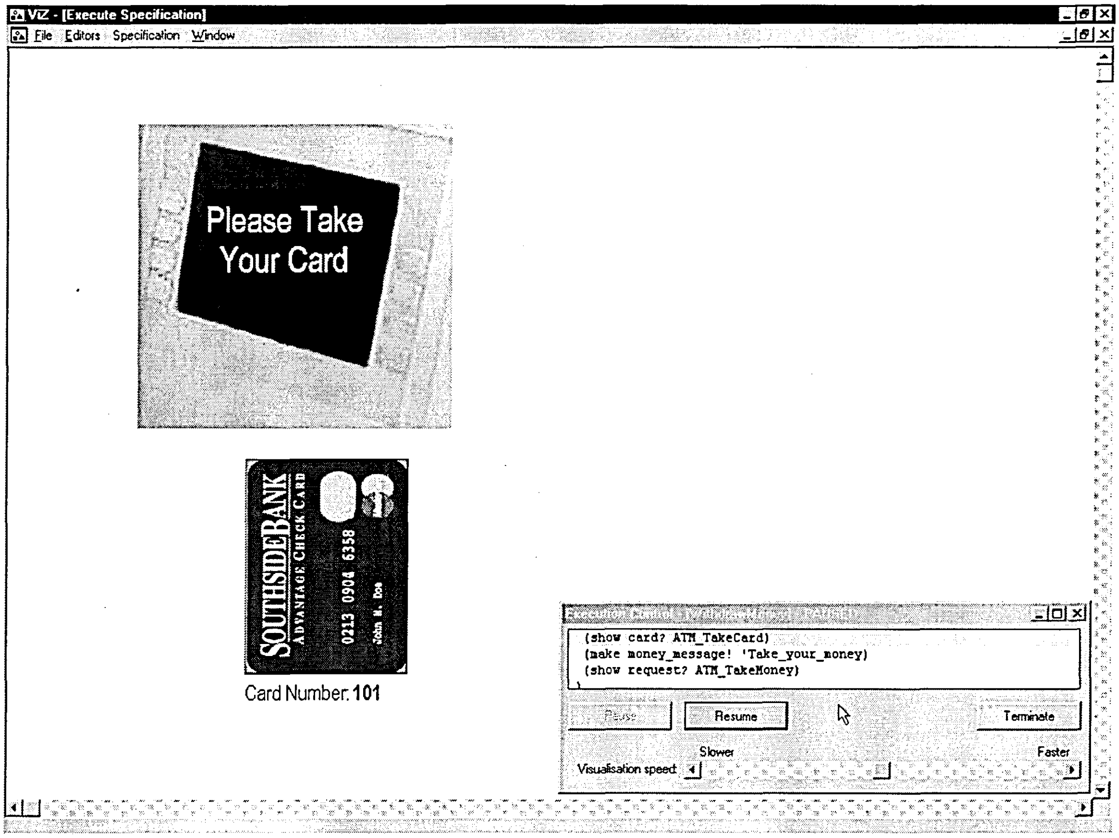
```
(lessorequal request? (applyz balance card?) ATM_ValidateAmount)
(make temp (- (applyz balance card?) request?))
(make balance' (override balance [#(card? temp)]))
(make available' (- available request?))
```

***Scene** 5. Entering the required amount of money.*


ViZ - [Execute Specification]
File Editors Specification Window
Amount OK
Pause
Resume
Terminate
Slower
Faster
Visualisation speed

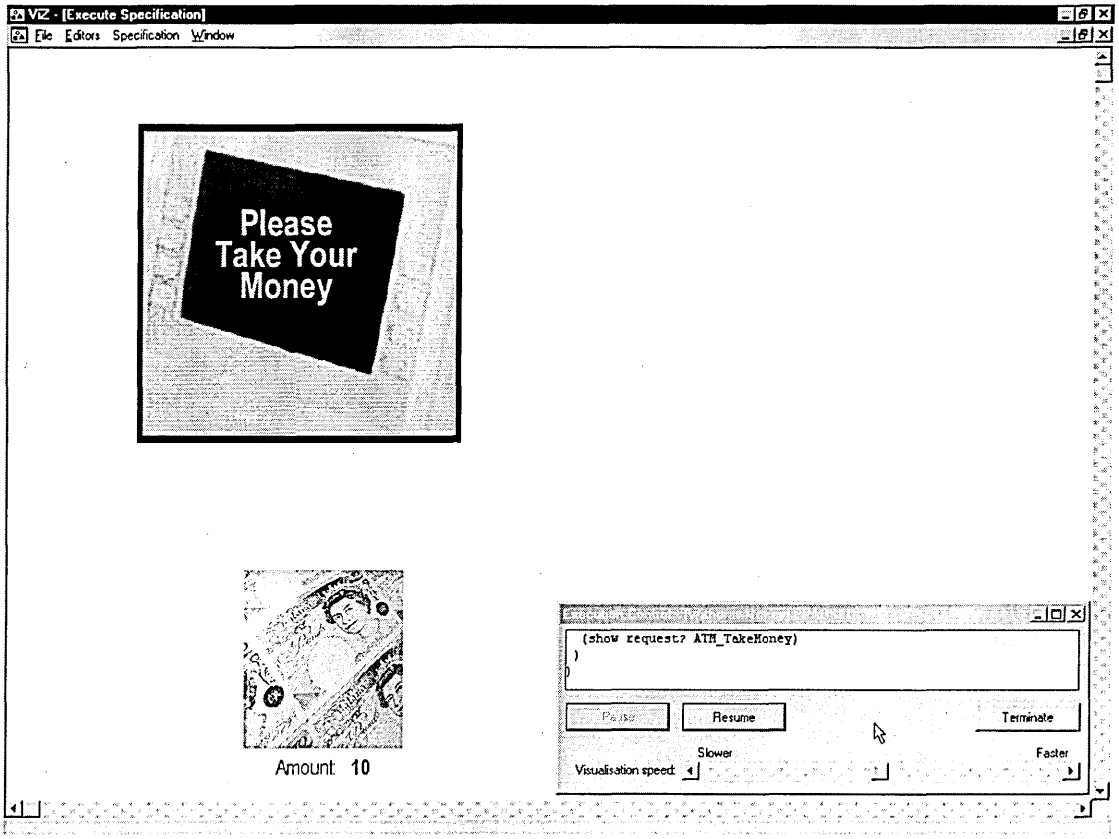
```
(lessorequal request? (applyz balance card?) ATM_ValidateAmount)
(make temp (- (applyz balance card?) request?))
(make balance' (override balance [#(card? temp)]))
(make available' (- available request?))
```

***Scene 6.** The amount is validated as being sufficient.*



```
(show card? ATM_TakeCard)
(make money_message! 'Take_your_money)
(show request? ATM_TakeMoney)
)
```

Scene 7. Informing the ATM customer to take their card.



```
 (show request? ATM_TakeMoney)
)
)
```

Scene 8. Informing the customer to take their money.

Appendix D – Email System Formal Specification

Appendix D presents the Z specification that models the email system used as the basis for the second case study (*Section 4.2*). Its origins lie in a specification for a complete electronic office system, of which the email system comprises a large component [Cohen86]. For brevity, only the salient points of the specification are presented, but where appropriate, the author's comments and notes have been retained.

$$
\begin{array}{l}
[\mathrm{NAME},\ \mathrm{USERID},\ \mathrm{DATE},\ \mathrm{TRAYID},\ \mathrm{MAILID},\ \mathrm{FREETEXT}] \\
\mathrm{BOOL} ::= \mathrm{Yes} \mid \mathrm{No} \\
\mathrm{ADMINID} ::= \mathrm{admin}
\end{array}
$$

email

$$
\begin{array}{ll}
\mathrm{deskof} & : \mathrm{USERID} \nrightarrow \mathrm{TRAYID} \\
\mathrm{inbox} & : \mathrm{TRAYID} \nrightarrow \mathbb{P}\ \mathrm{MAILID} \\
\mathrm{pending} & : \mathrm{TRAYID} \nrightarrow \mathbb{P}\ \mathrm{MAILID} \\
\mathrm{outbox} & : \mathrm{TRAYID} \nrightarrow \mathbb{P}\ \mathrm{MAILID} \\
\mathrm{direct} & : \mathrm{USERID} \nrightarrow (\mathrm{NAME} \nrightarrow \mathrm{USERID}) \\
\mathrm{mailitem} & : \mathbb{P}\mathrm{MAILID} \\
\mathrm{mailto} & : \mathrm{MAILID} \nrightarrow \mathrm{seq}\ \mathrm{TRAYID} \\
\mathrm{mailcc} & : \mathrm{MAILID} \nrightarrow \mathrm{seq}\ \mathrm{TRAYID} \\
\mathrm{mailfrom} & : \mathrm{MAILID} \nrightarrow \mathrm{NAME} \\
\mathrm{mailsubject} & : \mathrm{MAILID} \nrightarrow \mathrm{FREETEXT} \\
\mathrm{mailreplyreq} & : \mathrm{MAILID} \nrightarrow \mathrm{BOOL} \\
\mathrm{mailrefs} & : \mathrm{MAILID} \nrightarrow \mathbb{P}\ \mathbb{N} \\
\mathrm{mailwhensent} & : \mathrm{MAILID} \nrightarrow \mathrm{DATE} \\
\mathrm{mailrefno} & : \mathrm{MAILID} \nrightarrow \mathbb{N} \\
\mathrm{mailbody} & : \mathrm{MAILID} \nrightarrow \mathrm{FREETEXT} \\
\mathrm{x} & : \mathrm{MAILID} \\
\mathrm{admin} & : \mathrm{ADMINID}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{dom\ deskof} = \mathrm{dom\ direct} \\
\forall\ u : \mathrm{USERID} \mid u \in \mathrm{dom\ deskof} \bullet \mathrm{ran}(\mathrm{direct}(u)) \subseteq \mathrm{dom\ deskof} \\
\mathrm{dom\ mailrefno} = \bigcup(\mathrm{ran\ inbox} \cup \mathrm{ran\ pending}) \\
\mathrm{dom\ mailwhensent} = \bigcup(\mathrm{ran\ inbox} \cup \mathrm{ran\ pending})
\end{array}
$$

$x = \bigcup(\text{ran inbox} \cup \text{ran pending})$

$\forall\ m1,m2 : MAILID \mid m1 \in x \wedge m2 \in x \bullet mailrefno(m1) = mailrefno(m2) \Rightarrow m1 = m2$

$\forall\ m : MAILID \mid m \in \bigcup(\text{ran outbox}) \bullet m \notin \text{dom mailrefno}$

$\text{dom inbox} = \text{ran deskof}$

$\text{dom pending} = \text{ran deskof}$

$\text{dom outbox} = \text{ran deskof}$

$\text{dom mailto} = mailitem$

$\text{dom mailcc} = mailitem$

$\forall\ m : MAILID \mid m \in \text{dom mailto} \bullet mailto(m) \neq \emptyset$

$\text{ran}(\bigcup(\text{ran mailto})) \subseteq \text{ran deskof}$

$\text{ran}(\bigcup(\text{ran mailcc})) \subseteq \text{ran deskof}$

$\text{dom mailfrom} = mailitem$

$\text{dom mailsubject} = mailitem$

$\text{dom mailreplyreq} = mailitem$

$\text{dom mailrefs} = mailitem$

$\text{dom mailbody} = mailitem$

POST

$\Delta email$

$uid? : USERID$

$m? : MAILID$

$to? : \text{seq } NAME$

$cc? : \text{seq } NAME$

$from? : NAME$

$subject? : FREETEXT$

$reply? : BOOL$

$refs? : \mathbb{N}$

$body? : FREETEXT$

$uid? \in \text{dom deskof}$

$m? \notin mailitem$

$to? \neq \emptyset$

$\text{ran } to? \subseteq \text{dom}(direct(uid?))$

$\text{ran } cc? \subseteq \text{dom}(direct(uid?))$

$mailitem' = mailitem \cup \{m?\}$

$mailto' = mailto \cup \{m? \mapsto squash(\{(k,p) : \mathbb{N} \times TRAYID \mid k \in \text{dom } to? \wedge p =$
$deskof(direct(invoker?)(to?(k)))\})\}$*/

$mailcc' = mailcc \cup \{m? \mapsto squash(\{(k,p) : \mathbb{N} \times TRAYID \mid k \in \text{dom } cc? \wedge p =$
$deskof(direct(invoker?)(cc?(k)))\})\}$*/

$mailfrom' = mailfrom \cup \{m? \mapsto from?\}$

$mailsubject' = mailsubject \cup \{m? \mapsto subject?\}$

$mailreplyreq' = mailreplyreq \cup \{m? \mapsto reply?\}$

$mailrefs' = mailrefs \cup \{m? \mapsto refs?\}$

$mailbody' = mailbody \cup \{m? \mapsto body?\}$

$outbox' = outbox \oplus \{deskof(uid?) \mapsto outbox(deskof(uid?)) \cup \{m?\}\}$

/*

POST takes a mail from the "pad" of the user (outside the system) and adds it to their outbox. This operation performs the following tasks:

Checks the mail id is not known
Checks that the TO list is not empty
Checks that those identified in the TO list are known aliases from the user's directory
Does the same for those identified on the CC list
Adds the mail item to the list of known mails in the system
Creates a new version of the mailto set with the new mail added and the TO list changed so that the intended recipients' trays are present instead of their names.
Creates a new version of the mailto set with the new mail added and the CC list changed so that the intended recipients' trays are present instead of their names.
Completes the rest of the mail details provided.

*/

COLLECT

$\Delta email$

uid?	:	USERID
refno?	:	$\mathbb{N}$
m	:	MAILID

$uid? \in dom\ deskof$

$refno? \in ran\ mailrefno$

$m = (mailrefno^{\sim})(refno?)$

$m \in inbox(deskof(uid?))$

$inbox' = inbox \oplus \{deskof(uid?) \mapsto inbox(deskof(uid?)) \setminus \{m\}\}$

/*

COLLECT allows a user to remove a mailitem from their inbox in order to preview it on their "PAD". As the composing pad is outside the system it is simply removed. The operation checks that the mail reference number is applicable and is in the inbox of the user. The mail is then removed from the inbox. */

```
┌READ──────────────────────────────────────
│ Ξemail
│ uid?            : USERID
│ refno?          : ℕ
│ m               : MAILID
├──────────────────────────────
│ uid? ∈ dom deskof
│ refno? ∈ ran mailrefno
│ m = (mailrefno~)(refno?)
│ m ∈ pending(deskof(uid?))
└──────────────────────────────────────
```

/*

READ is an operation which is used to allow a user to check the contents of a pending mail item (preview). It has to stay in the pending tray. This operation simply checks that the user can do this. */

```
┌LIST──────────────────────────────────────
│ Ξemail
│ invoker?            : USERID
│ mailids!            : ℙMAILID
│ mailsubjects!       : MAILID ⇸ FREETEXT
│ mailfroms!          : MAILID ⇸ NAME
│ mailwhensents!      : MAILID ⇸ DATE
│ mailrefnos!         : MAILID ⇸ ℕ
├──────────────────────────────
│ invoker? ∈ dom deskof
│ mailids! = inbox(deskof(invoker?))
│ mailsubjects! = mailids! ◁ mailsubject
│ mailfroms! = mailids! ◁ mailfrom
│ mailwhensents! = mailids! ◁ mailwhensent
│ mailrefnos! = mailids! ◁ mailrefno
└──────────────────────────────────────
```

/* LIST list all mails in the inbox of the user in question */

```
┌LISTP──────────────────────────────────────────
│ Ξemail
│ invoker?          : USERID
│ mailids!          : ℙMAILID
│ mailsubjects!     : MAILID ⇸ FREETEXT
│ mailfroms!        : MAILID ⇸ NAME
│ mailwhensents!    : MAILID ⇸ DATE
│ mailrefnos!       : MAILID ⇸ ℕ
├───────────────────────────────────
│ invoker? ∈ dom deskof
│ mailids! = pending(deskof(invoker?))
│ mailsubjects! = mailids! ◁ mailsubject
│ mailfroms! = mailids! ◁ mailfrom
│ mailwhensents! = mailids! ◁ mailwhensent
│ mailrefnos! = mailids! ◁ mailrefno
└──────────────────────────────────────────
```

/* LISTP list all mails in the pending tray of the user in question */

```
┌SEND───────────────────────────────────────────
│ Δemail
│ invoker?          : USERID
│ mail?             : MAILID
│ now?              : DATE
│ ref?              : ℕ
│ pendingmails      : MAILID
│ outgoingto        : MAILID ⇸ seq TRAYID
│ outgoingcc        : MAILID ⇸ seq TRAYID
│ pendingsrefs      : MAILID ⇸ ℕ
│ outgoingrefs      : MAILID ⇸ ℙ ℕ
│ outgoingreplies   : ℙMAILID
│ outgoingrepliesreq    : ℙMAILID
│ f                 : TRAYID ⇸ ℙ MAILID
│ g                 : TRAYID ⇸ ℙ MAILID
│ h                 : TRAYID ⇸ ℙ MAILID
│ trays             : ℙTRAYID
│ trayscc           : TRAYID
├───────────────────────────────────
│
```

$invoker? \in \mathrm{dom}\ deskof$

$\#(outbox(deskof(invoker?))) > 0$

$mail? \in outbox(deskof(invoker?))$

$ref? \notin \mathrm{ran}\ mailrefno$

$pendingmails = pending(deskof(invoker?))$

$outgoingto = \{mail?\} \lhd mailto$

$outgoingcc = \{(mi,x) : MAILID \times \mathrm{seq}\ TRAYID \mid mi \in \mathrm{dom}(\{mail?\} \lhd mailcc) \wedge (x =$
$\quad mailcc(mi) \mid \quad \wedge x \neq \varnothing)\}*/$

$pendingsrefs = pendingmails \lhd mailrefno$

$outgoingrefs = \{mail?\} \lhd mailrefs$

$outgoingreplies = \{m : MAILID \mid m \in pendingmails \wedge pendingsrefs(m) \in \bigcup(\mathrm{ran}$
$\quad outgoingrefs)\}$

$outgoingrepliesreq = \{m : MAILID \mid m \in \{mail?\} \wedge mailreplyreq(m) = Yes\}$

$outbox' = outbox \oplus \{deskof(invoker?) \mapsto outbox(deskof(invoker?)) \setminus \{mail?\}\}$

$trays = \mathrm{ran}(outgoingto(mail?))$

$f = \{(tr, x) : TRAYID \times \mathbb{P}\ MAILID \mid tr \in trays \wedge x = inbox(tr) \cup \{mail?\}\}*/$

```
if mail? ∈ outgoingrepliesreq
      then g = {(tr,x) : TRAYID × ℙ MAILID | tr ∈ trays ∧ x = pending(tr) ∪ {mail?}}
else g = ∅
if #(outgoingcc) > 0 then
      (trayscc = ran outgoingcc(mail?) ∧
      h = {(tr, x) : TRAYID × ℙ MAILID | tr ∈ trayscc ∧ x = inbox(tr) ∪ {mail?}}
else
      h = ∅
```

$pending' = pending \oplus g \oplus \{deskof(invoker?) \mapsto pending(deskof(invoker?)) \setminus outgoingreplies\}$

$inbox' = inbox \oplus f \oplus h$

$mailwhensent' = mailwhensent \cup \{mail? \mapsto now?\}$

$mailrefno' = mailrefno \cup \{mail? \mapsto ref?\}$

/*

SEND allows a user to send a given mail item in their outbox to the intended recipients. The operation does the following tasks:

The system checks the user is known and that their outbox is not empty.

It checks the given mail item is in the outbox and that the reference number to be assigned to it is not already allocated.

It removes the mail from the outbox of the user

It determines the trays that the mail is going to.

It creates a temporary variable f which is the contents of all inboxes in which the

```
mail    has been copied into the trays identified above.
        If a reply is required a temporary variable g is created containing all pending trays
in      which the mail has been copied into the trays identified above.
        If no reply is required, g is empty.
        If the CC list of the mail is not empty then it determines which trays require a
        copy.
        It creates a variable h containing the inboxes in which the mail has been copied
        into the trays identified from the CC list.
        If the CC list is empty h is assigned the empty set.
        The pending trays are overridden with the variable g (recipients on the TO list for
                a mail requiring a reply) and any mails in the user's pending tray, which
        were identified as being replied to with the mail in question, are also removed from
        the pending trays.
        The inboxes are overridden with the variables f and h to reflect that the mails
        have been sent to those on TO and CC lists.
        The date stamp for the mail is created
        The reference number for the mail is created.
*/
```

Appendix E – WLMS Formal Specification

Appendix E presents the Z specification that models the WLMS [VanShouwen96] that is used as the basis for the third case study (*Section 4.3*). The specification is presented in its entirety, including the author's preamble and comments.

Case Study : Water Level Monitoring System in Z

From : A. Van Shouwen

Date : 20th September 1996

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 transferring 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.

```
/* Define Types to be used within the specification */

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
HEATERSWITCHTYPE        ::= open | closed
SHUTDOWNSIGNALTYPE      ::= go | stop
ALARMTYPE               ::= silent | audible

/* Now define some variables of these types, grouped by function */

┌ButtonTimes──────────────────────────────────
│ resetButtonTime       : TIME
│ selftestButtonTime    : TIME
│
└─────────────────────────────────────────────

┌Modes────────────────────────────────────────
│ operatingMode         : OPERATINGMODETYPE
│ failureMode           : FAILUREMODETYPE
│
└─────────────────────────────────────────────

/* Now we define the data structures associated with our model */

┌StoredVar────────────────────────────────────
│ alarm                 : ALARMTYPE
│ shutdownSignal        : SHUTDOWNSIGNALTYPE
│
└─────────────────────────────────────────────
```

```
┌StoredData──────────────────────────────────
│ time                  : TIME
│ timeInMode            : TIME
│ watchDogTime          : TIME
│ Modes
│ ButtonTimes
└────────────────────────────────────────────
```

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

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

/* Now we introduce the constant global values associated with the WLMS */

```
┌
│ 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
└
```

```
┌PumpControl──────────────────────────────────────────
│ MonitorVar?
│ ΔModes
│ ΔStoredVar
├──────────────────────────────────────────
│ (operatingMode = operating ∧ failureMode = allok ∧ (¬(resetButton? = pressed)) ⇒
│       shutdownSignal' = go)
│ (operatingMode = operating ∧ failureMode = allok ∧ resetButton? = pressed ⇒
│       shutdownSignal' = shutdownSignal)
│ (operatingMode = shutdown ∧ failureMode = allok ⇒
│       shutdownSignal' = shutdownSignal)
│ (operatingMode ∈ {standby,test} ∧ failureMode = allok ⇒ shutdownSignal' = stop)
│ (failureMode ∈ {badlevdev, hardfail} ⇒ shutdownSignal = stop)
└──────────────────────────────────────────
```

```
AlarmControl
  MonitorVar?
  waterlevel          : LEVEL
  ΔStoredData
  ΔStoredVar
  ----------------------------------------
  powerNow? = on
  (operatingMode ∈ {shutdown,operating} ∧ failureMode = allok ∧
        (¬(lowWaterLimit < waterlevel <highWaterLimit)) ⇒
        alarm' = audible)
  (operatingMode = operating ∧ failureMode = allok ∧ timeInMode = 0  ∧
        (lowWaterLimit < waterlevel <highWaterLimit) ⇒
        alarm' = silent)
  (operatingMode = operating ∧ failureMode = allok ∧ (¬(timeInMode = 0))  ∧
        (lowWaterLimit < waterlevel <highWaterLimit) ⇒
        alarm' = alarm)
  (operatingMode = shutdown ∧ failureMode = allok ∧
        (lowWaterLimit < waterlevel <highWaterLimit) ⇒
        alarm' = alarm)
  (operatingMode = standby ∧ failureMode = allok ⇒ alarm' = alarm)
  (operatingMode = test ∧ failureMode = allok ⇒
        alarm' = (if 0 ≤ timeInMode < maxAlarmTime then audible else silent))
  (failureMode = badlevdev ∧ timeInMode = 0 ⇒ alarm' = audible)
  (failureMode = badlevdev ∧ (¬(timeInMode = 0)) ⇒ alarm' = alarm)
```

```
PumpEnvironment
  powerNow?           : ONOFFTYPE
  ControlSignals!
  StoredVar'
  ----------------------------------------
 (powerNow? = on ∧ shutdownSignal' = go ∧ watchdog! = operate ⇒
        pumpSwitch! = closed)
 (powerNow? = off ∨ shutdownSignal' = stop ∨ watchdog! = shut ⇒
        pumpSwitch! = open)
```

```
┌WasOperating──────────────────────────
│ MonitorVar?
│ ΔModes
│ ButtonTimes
│ waterlevel            : LEVEL
├──────────────────────────
│ operatingMode = operating
│ operatingMode' =
│       if selftestButton? ≠ pressed ∧ (lowWaterLimit < waterlevel < highWaterLimit)
│       then shutdown
│       else if selftestButton? = pressed ∧ selftestButtonTime ≥ maxSelftestDelay
│       then test
│       else operating
└──────────────────────────────
```

```
┌WasShutdown──────────────────────────
│ MonitorVar?
│ ΔStoredData
│ waterlevel            : LEVEL
├──────────────────────────
│ operatingMode = shutdown
│ operatingMode' =
│       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
└──────────────────────────────
```

WasinStandby

MonitorVar?

ΔModes

ButtonTimes

waterlevel : LEVEL

---

operatingMode = standby

resetButtonTime $\geq$ maxResetDelay $\wedge$

(lowWaterLimit + hysteresis < waterlevel < highWaterLimit - hysteresis) $\Rightarrow$

operatingMode' = operating

selftestButton? $\neq$ pressed $\wedge$ selftestButtonTime $\geq$ maxSelftestDelay $\Rightarrow$

operatingMode' = test

(($\neg$(resetButtonTime $\geq$ maxResetDelay)) $\wedge$

(lowWaterLimit + hysteresis < waterlevel < highWaterLimit - hysteresis) $\wedge$

($\neg$(selftestButton? $\neq$ pressed $\wedge$ selftestButtonTime $\geq$ maxSelftestDelay)) $\Rightarrow$

operatingMode' = standby)

WasinTest

ΔStoredData

---

operatingMode = test

operatingMode' =

if timeInMode $\geq$ maxTestDelay

then standby

else test

WasAllok

MonitorVar?

ΔModes

levelDevice : DEVICETYPE

controlUnit : DEVICETYPE

---

failureMode = allok

failureMode' =

    if levelDevice = failed $\wedge$ controlUnit = ok $\wedge$ timeDevice? = ok

    then badlevdev

    else if controlUnit = failed $\vee$ timeDevice? = failed

    then hardfail

    else allok

WasBadLevdev

MonitorVar?

ΔModes

controlUnit : DEVICETYPE

---

failureMode = badlevdev

failureMode' =

    if controlUnit = failed $\vee$ timeDevice? = failed

    then hardfail

    else badlevdev

WasHardFail

ΔModes

---

failureMode = hardfail

failureMode' = hardfail

GetNextMode $\widehat{=}$ (WasOperating $\vee$ WasShutdown $\vee$ WasinStandby $\vee$ WasinTest) $\wedge$
(WasAllok $\vee$ WasBadLevdev $\vee$ WasHardFail)

CheckButtonTimes

MonitorVar?

ΔButtonTimes

step : TIME

$resetButton? = pressed \Rightarrow resetButtonTime' = resetButtonTime + step$

$resetButton? = released \Rightarrow resetButtonTime' = 0$

$selftestButton? = pressed \Rightarrow selftestButtonTime' = selftestButtonTime + step$

$selftestButton? = released \Rightarrow selftestButtonTime' = 0$

GetInmodeTime

ΔStoredData

step : TIME

$(operatingMode = operatingMode' \wedge timeInMode' = timeInMode + step) \vee$

$(operatingMode \neq operatingMode' \wedge timeInMode' = 0)$

CheckControlUnit

MonitorVar?

watchdog! : WATCHDOGTYPE

controlUnit : DEVICETYPE

$memory? = ok \wedge watchdog! = operate \Rightarrow controlUnit = ok$

$memory? = failed \vee watchdog! \in \{uninit, shut\} \Rightarrow controlUnit = failed$

CheckLevelDevice

MonitorVar?

levelDevice : DEVICETYPE

$diffPress? \in \{0,255\} \Rightarrow levelDevice = failed$

$\neg(diffPress? \in \{0,255\}) \Rightarrow levelDevice = ok$

```
┌UpdateStoredData──────────────────────────
│ MonitorVar?
│ ΔStoredData
│ ΔStoredVar
│ ControlSignals!
│ controlUnit        : DEVICETYPE
│ waterlevel         : LEVEL
│ step               : TIME
│ levelDevice        : DEVICETYPE
├──────────────────────────────
│ step = timeNow? - time
│ CheckControlUnit
│ CheckLevelDevice
│ time' = timeNow?
│ GetInmodeTime
│ watchDogTime' = step
│ GetNextMode
│ CheckButtonTimes
└──────────────────────────────────
```

```
┌GetWaterLevel──────────────────────────
│ diffPress?         : BYTE
│ waterlevel         : LEVEL
├──────────────────────────────
│ (let level == levelLowerCal +
│      ((diffPress? * 103803 - 485010) * (levelUpperCal - levelLowerCal)) div 2550000
│           • diffPress? = 255 ∧ waterlevel < levelLowerCal ∨
│      diffPress? = 0 ∧ waterlevel > levelUpperCal ∨
│      1 ≤ diffPress? ≤ 255 ∧ waterlevel =
│           if level < levelLowerCal then levelLowerCal
│           else if level > levelUpperCal then levelUpperCal
│           else level)
└──────────────────────────────────
```

```
┌CheckTimer──────────────────────────
│ watchDogTime               : TIME
│ watchdog!          : WATCHDOGTYPE
├──────────────────────────────
│ watchdog! = (if watchDogTime < watchdogtimeout then operate else shut)
└──────────────────────────────────
```

Initialise

StoredData

StoredVar

$0 \leq initTime \leq 5000$

alarm = silent

shutdownSignal = stop

timeInMode = 0

watchDogTime = 0

resetButtonTime = 0

selftestButtonTime = 0

operatingMode = standby

failureMode = allok

$0 \leq time \leq initTime$

HeaterEnvironment

MonitorVar?

ΔStoredData

ΔStoredVar

ControlSignals!

```
if pumpSwitch! = open ∧ alarm = audible
then
        heaterSwitch! = open
else
        heaterSwitch! = closed
```

```
┌NormalOperation──────────────────────────────
│ MonitorVar?
│ ΔStoredData
│ ΔStoredVar
│ ControlSignals!
│ controlUnit          : DEVICETYPE
│ waterlevel           : LEVEL
│ step                 : TIME
│ levelDevice          : DEVICETYPE
├─────────────────────────────────────
│ PumpControl
│ GetWaterLevel
│ CheckTimer
│ PumpControl
│ AlarmControl
│ PumpEnvironment
│ HeaterEnvironment
│ UpdateStoredData
│ shutdownSignal! = shutdownSignal'
│ alarm! = alarm'
└─────────────────────────────────────────────
```