A Graphical Environment Supporting the Algebraic Specification of Abstract Data Types

Kevin William Waite, B.Sc.

Submitted for the degree of
Doctor of Philosophy

Department of Computing Science
University of Glasgow

1990

Table of Contents

Table of Figures

Acknowledgements

I would like to thank my supervisor Ray Welland for his support, ideas and "gentle" cajoling, and particularly for providing detailed and constructive criticism of early drafts of this thesis. On a personal level, I was pleased that Ray's coaxing even resorted to bribes such as *"I'll find you a Wood Warbler if you finish chapter such-and-such"*. Now that this thesis has been finished, I hope he understands when I no longer let him beat me at badminton. I would also like to thank Alistair Kilgour (now at Heriot-Watt University) for his supervision and encouragement in the early stages of my study at Glasgow.

As a novice in interface evaluation I am grateful for the help I received in designing, conducting and analysing the results of the VISAGE evaluation. Steve Draper and Keith Oatley introduced me to the techniques and principles involved. Rex Hartson, a visitor to the department from Virginia Polytechnic Institute, suggested improvements to the VISAGE tutorial and evaluation task. I would also like to thank the numerous evaluation "volunteers" who gave me my first experience of evaluation, although I didn't enjoy it at the time.

I would like to thank my colleagues in the department, particularly the Graphics and HCI research group, for providing such an encouraging and supportive environment. In particular, Tunde Cockshot helped with the design of the VISAGE icons, while Phil Gray helped put my work in perspective. The initial suggestion of looking at algebraic specification came from Kieran Clenaghan, and who along with Muffy Thomas, gave useful feedback on the specification aspects of this thesis and earlier papers.

I would like to thank the members of the DRUID research group, and in particular their over-worked research assistant, for allowing me to use "epi". I am especially grateful to the DRUIDs for introducing me to their high priestess.

I am grateful to the Curators' department of the London Transport Museum for tracking down information on the history and evolution of the Underground Map.

On a personal side, I am indebted to my fiancée Catherine Wood for her ceaseless support and encouragement. In her rôle as unpaid supervisor, Cathy willingly discussed dry technical issues and helped find the numerous rough edges of early drafts of this thesis. I only hope I can provide the same support in return. I would also like to thank my family for supporting their perpetual student.

I would also like the acknowledge the financial support of the Science and Engineering Research Council (SERC) of the United Kingdom.

Abstract

Abstract Data Types (ADTs) are a powerful conceptual and practical device for building high-quality software because of the way they can describe objects whilst hiding the details of how they are represented within a computer. In order to implement ADTs correctly, it is first necessary to precisely describe their properties and behaviour, typically within a mathematical framework such as algebraic specification. These techniques are no longer merely research topics but are now tools used by software practitioners. Unfortunately, the high level of mathematical sophistication required to exploit these methods has made them unattractive to a large portion of their intended audience. This thesis investigates the use of computer graphics as a way of making the formal specification of ADTs more palatable.

Computer graphics technology has recently been explored as a way of making computer programs more understandable by revealing aspects of their structure and run-time behaviour that are usually hidden in textual representations. These graphical techniques can also be used to create and edit programs. Although such visualisation techniques have been incorporated into tools supporting several phases of software development, a survey presented in this thesis of existing systems reveals that their application to supporting the formal specification of ADTs has so far been ignored.

This thesis describes the development of a prototype tool (called VISAGE) for visualising and visually programming formally-specified ADTs. VISAGE uses a synchronised combination of textual and graphical views to illustrate the various facets of an ADT's structure and behaviour. The graphical views use both static and dynamic representations developed specifically for this domain. VISAGE's visual programming facility has powerful mechanisms for creating and manipulating entire structures (as well as their components) that make it at least comparable with textual methods.

In recognition of the importance of examples as a way of illustrating abstract concepts, VISAGE provides a dedicated tool (called the PLAYPEN) that allows the creation of example data by the user. These data can then be transformed by the operations belonging to the ADT with the result shown by means of a dynamic, graphical display.

An evaluation of VISAGE was conducted in order to detect any improvement in subjects' performance, confidence and understanding of ADT specifications. The subjects were asked to perform a set of simple specification tasks with some using VISAGE and the others using manual techniques to act as a control. An analysis of the results shows a distinct positive reaction from the VISAGE group that was completely absent in the control group thereby supporting the thesis that the algebraic specification of ADTs can be made more accessible and palatable though the use of computer graphic techniques.

Declaration

The material presented in this thesis is entirely the result of my own independent research carried out at the Department of Computing Science, University of Glasgow under the supervision of Mr. Ray Welland and formerly, Dr. Alistair Kilgour (now Professor at Heriot-Watt University, Edinburgh). Any published or unpublished material used by me has been given full acknowledgement in the text.

Trademarks

A number of trademarks are used in this thesis and for brevity are declared once here as follows but apply throughout the thesis:

- *Smalltalk-80* is a trademark of ParcPlace Systems.
- *Sun-3, SunView* and *Sun Workstation* are trademarks of Sun Microsystems, Inc.
- *UNIX* and C++ are trademarks of AT&T Bell Laboratories.
- *Macintosh, MacPaint* and are trademarks of Apple Computer, Inc.
- *MacDraw* is a trademark of the Claris Corporation.
- *Ada* is a trademark of the US Department of Defense.

Typographic Conventions

- Names of systems are shown in small capitals, e.g. VISAGE.
- `Algebraic specifications are shown in Courier 10 point.`
- ADT Names given in Helvetica 10 point, e.g. stack.
- VISAGE commands are shown in Helvetica 12 point, e.g. close.
- *"Direct quotations are shown in italics."*

Format of the Figures

This thesis contains many figures illustrating the graphical output generated by the VISAGE system. Although screen dumps of actual displays would have been preferred, these images often have poor contrast, making them difficult to reproduce. For this reason they have been replaced by hand-drawn figures that retain the character of the original while improving their clarity. The figures were created using the MacDraw graphics editor.

Gender

Male pronouns have been used in this thesis to refer to the androgynous "user" in order to smooth the flow of the text rather than imply any sexual bias.

Chapter One
Introduction

The construction of contemporary software bears a marked likeness to the construction of European cathedrals in the Middle Ages. Both were the most advanced projects of their day, demanding enormous human and financial resources. Both were developed by artisans who had acquired their skills by experience and from watching a master craftsman. The projects were usually over budget and behind schedule. Although in both cases dependable, solid structures were sometimes produced, in many cases the constructors found their creations falling on top of them. Just as the construction of cathedrals only became reliable with the adoption of the disciplines of what is now civil engineering, only through the adoption of disciplined methodologies can reliable and trusted software be built. The need for software to become a branch of engineering rather than a black art has been eloquently argued by Hoare (1982), among others. It is now widely recognised that robust and dependable software structures can only be built if they are firmly embedded in sound mathematical foundations.

Recently, such foundations have started to be incorporated into tools and formal methods that the practising software engineer (as opposed to the theoretical computer scientist) can apply to problems. One such foundation used as the case study in this thesis is concept of the *Abstract Data Type* (ADT). This is a mechanism for categorising objects according to their behaviour and the interface they present to other objects. ADTs were chosen as the case study because of their importance in software engineering and because they have a neatly circumscribed formal basis which can be understood relatively easily. ADTs have now become a weapon in the general programming armoury appearing as *classes* in fashionable object-oriented languages such as Smalltalk and C++, and *packages* in more conventional ones such as Ada.

Abstract data types are important in the development of large software systems as they allow a large project to be partitioned into manageable chunks which can be tackled by different programmers. This is possible because the interface of each ADT, which describes its behaviour and outward appearance, can be stated precisely, before implementation begins. Given such a specification, programmers can make use of chunks being implemented independently by other programmers, confident that these chunks will behave as intended. The well-defined interfaces of the ADTs ensures that the implemented chunks can later be combined to form the final system without any problems due to hidden dependencies between chunks. These chunks may even be re-used in other software systems: a programmer could select an ADT from a software re-use library by browsing a catalogue of specifications and selecting the one that matches the given requirements.

To ensure the necessary precision in the description of an ADT's interface its specification should be written in a language with precisely defined semantics (e.g. mathematics); in this case the description is called a *formal specification*. To summarise: ADTs are a powerful design tool as they localise functionality, hide implementation details, and restrict communication between chunks to that permitted by a defined interface.

Although these new mathematically-based techniques provide the necessary formal framework for the development of provably correct software they currently suffer from a number of drawbacks. Firstly, the tools and techniques are currently not equal to the task of describing software of the size currently being developed: for example, there is an enormous scaling problem in bridging the gap between verifying the correctness of the (white rat) factorial program and the correctness of the avionics software of the US Space Shuttle (Spector and Gifford, 1984).

Although mathematical techniques still need to be developed to handle certain classes of software (e.g. those involving floating-point arithmetic or processing in real-time), a major reason why formal methods have not been widely adopted is the lack of mechanical support. It is not unusual for the proof of a program's correctness to be many times longer than the program itself. The question of correctness now passes onto the proof: if it has been derived manually then there is considerable risk that it contains errors itself. Clearly, mechanical assistance is required to handle the tedious administrative and rote aspects involved in checking the correctness of programs, as well as checking the legality of the steps involved. These problems of scale and mechanical support of correctness proving are *not* addressed by this thesis.

Another problem with formal methods is at a more human level. Programmers raised in the craft style typically do not have the mathematical skills and intuitions required to exploit these new techniques. Although the trend in Computer Science teaching is towards a more formal approach, graduates of such programmes still constitute the minority of programmers, most of whom have acquired their skills the traditional way. Many practising programmers, while appreciating the potential benefits of a disciplined approach to software development, are intimidated by the esoteric mathematical notations typically adopted by these techniques. Clearly, a major, imaginative education (and re-education) programme is required if we are to meet the demand for people schooled in the techniques of rigorous software development. Such a programme needs to present the techniques in a more palatable way, appealing to novices' intuitions and experiences.

1.1 The Aims of this Thesis

Having identified the need for a more palatable introduction to ADTs and their formal specification, the goal becomes one of finding a pedagogical device that can present the necessary machinery in an appealing, yet rigorous, form. It is the main aim of this thesis

to investigate one promising approach to making ADT specifications more accessible, namely its integration within an interactive computer graphics environment.

The thesis argues that interactive computer graphics is a natural vehicle for introducing novices to formally-specified ADTs. The reasoning behind this argument is based on the observation that the textual notations used in specification and programming languages are considerably divorced from the mental representations that psychological study has shown to be used by most people when thinking. As figure 1.1 shows, the conventional textual (or symbolic) representation is closer to the computer's perspective than the user's because it is simpler to implement, heedless of the fact that many potential users find it incomprehensible. An extreme example of a representation biased towards the machine is assembly language with its rigid formats and obscure acronyms.

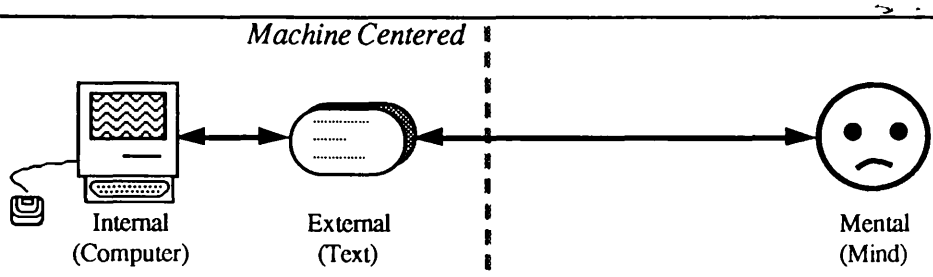


Figure 1.1: Traditional, Machine-Centred Representation

Such notations require the user to invest considerable effort in bridging the gap between how they think about a problem and the notation used to express its solution. One way of making formal specification more accessible to novices is by bringing the notations closer to the representations used in the mind, as shown in figure 1.2.

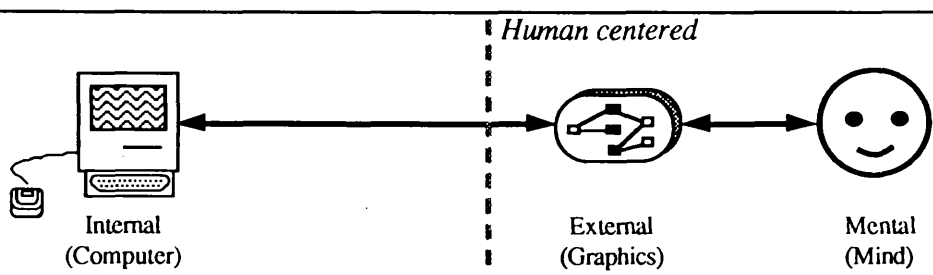


Figure 1.2: Proposed, Human-Centred Representation

This is the path taken by the evolution of traditional programming languages from primitive machine languages to domain-specific high-level languages. Each generation tries to provide an abstract computational model nearer the mental abstractions used by the programmer. For example, by providing numbers and functions, (even) FORTRAN provides a model of computation that is closer to an engineer's thinking than that of the underlying machine instructions with its bit patterns and machine operations. It has been

argued (e.g. by Goguen, 1986) that (executable) specification languages, such as OBJ, are the next generation in this evolutionary path. Unfortunately, psychological evidence (such as by Hadamard, 1945) suggests that the highly symbolic abstractions employed in languages such as OBJ have been developed for reasons of mathematical convenience rather than being conducive to human understanding. Despite the insatiable drive for even greater abstraction, the gap between mental and machine representations remains wide.

Instead of trying to evolve higher-level languages, an alternative assault on the problem has concentrated on using interactive computer graphics to produce representations of software entities that are nearer users' mental models. This use of graphics to describe various properties of an existing program has been termed *program visualisation.* When this machinery has been used to let the programmer create or manipulate the program in the first place then it is called *visual programming.* These terms are discussed in greater detail in subsequent chapters.

Advances in hardware technology provide the opportunity to explore novel approaches to developing software that were simply impossible only a few years ago. High-resolution displays allow the simultaneous display of multiple representations of the same data using a combination of text and graphics. It is this ability to use graphics that makes these hardware developments particularly exciting. When combined with a pointing device (such as a mouse), such hardware offers the potential for rich dialogues between human and computer. This is in complete contrast to the separated monologues that characterised interactive software before the introduction of such hardware. Psychological evidence that mental representations are rarely symbolic but often visual, suggests that a graphical language may allow the development of object representations that are closer to users' mental representations than would ever be possible using text alone.

Despite the evidence for non-textual modes of thinking, it is unlikely that programmers whose education has been dominated by textual and symbolic manipulation will switch completely to graphical means of specifying and developing software. It is unrealistic to expect to overcome the momentum of text-based methodologies in software development, at least in the short term. Moreover, such a revolution would be failing to exploit the full potential afforded by the new technology. Rather than simply replace a text-based representation with a graphical one, attempts should be made to create a synergy whereby each representation provides an insight into an aspect of a software artifact that is not provided by the other. For the first time, mechanised programming tools can use the plurality of notations that have previously been available only as paper design aids. Now it is possible for a programmer to use the representation or interaction technique that is best suited to the task at hand rather than being forced into using an inappropriate or inefficient one. For example, the decomposition of modules in a program may be best described graphically whereas at the atomic level of individual statements, a textual approach may be

easier. Only recently have programming tools supported mixed representations and interaction styles.

Educators, such as Papert (1980), have long known that the use of examples produces a marked improvement on the ease and speed at which new concepts can be learned, particularly if these concepts are of an abstract nature. The active manipulation of examples is known to be a much more potent learning device that the passive absorption of information. Recently, software designed for use by novices has used examples to illustrate some aspect of their operation. This has been taken to its extreme by a variety of novel, experimental computer systems whose computational model is the manipulation of examples of the data that the eventual program will manipulate. Such systems are discussed in detail in chapter three.

In order to investigate whether the application of computer graphics techniques to the formal specification of ADTs would actually be of benefit, a prototype system was developed, and used to experiment with various approaches. For ease of referral, this system was named VISAGE. A presentation of the evolution of VISAGE's design, together with an account of the experiences of building and then evaluating VISAGE, comprises the bulk of the thesis.

1.2 Contributions of this Thesis

The primary contributions of the VISAGE system are:

- The application of computer graphics techniques to formal specification using ADT specification as the case study.
- The development of new graphical representations for visualising the interface an ADT presents to its clients (syntax), its behaviour (semantics), and its relationships to other ADTs.
- The development of mechanisms that allow the creation and manipulation of new ADT specifications entirely within a graphical environment. Specialised interaction techniques have been developed to make this approach efficient.
- A demonstration, by evaluation of VISAGE, that users with little or no experience of algebraic specification can quickly produce specifications of simple but interesting ADTs with confidence in their correctness. Users generally enjoyed using VISAGE and would like to have such a tool for learning to specify ADTs.
- The provision of a graphical environment for rewriting terms to demonstrate how changes in the specifications' equations affect the behaviour of an ADT's operations.

Lower-level contributions include:

- The development of a combined help and commenting mechanism that juxtaposes user-supplied comments with a system-generated, context-sensitive description of any displayed object.
- A demonstration that multiple, synchronised views of a complex entity are a natural way of understanding its different facets, and that a combination of textual and graphical representations seems the most promising way of achieving this.
- An evaluation of the obscurities and complexities of Smalltalk's Model-View-Controller framework.

Although this thesis combines algebraic specification and visualisation techniques, its main contribution is in the latter domain. This thesis contributes nothing to algebraic specification per se, leaving such work to the theorists. Because the rôle of algebraic specification in this thesis is merely as a well-behaved subject only rudimentary aspects of the technique are needed. However, more advanced aspects are considered in the *Future Work* section in chapter eight.

1.3 Thesis Organisation

In many ways a thesis is an inadequate medium for reporting research because of the way it imposes a sanitised and often artificial order on the presentation of results. The tangled paths of exploration and backtracking are coerced into a strictly linear format that often highlights only the landmarks of success along the route. In this thesis an attempt has been made to also point out the dead ends and wrong turnings in the belief that these are perhaps of equal use in guiding any future research that may wander along this path.

As this thesis tries to bring together two previously disparate domains it is necessary to introduce each by presenting a survey of the work in the area at a level appropriate to the needs of the thesis. Chapter two starts by introducing ADTs from a formal perspective with particular regard to their benefits as a powerful and useful abstraction mechanism for the development of reliable, flexible software. Having discussed the engineering benefits of ADTs, the chapter moves on to their specification with a comparison of the various techniques available, emphasizing those that allow formal reasoning about an ADT's behaviour and the correctness of an implementation. These *formal specifications* are then placed in context within a simplified software life-cycle. This is followed by an introduction to the most widely used formal specification technique for ADTs, namely *algebraic specification*. The central points of the technique are presented by means of a small but representative example. This section also defines the associated terminology adopted for the rest of the thesis. We then examine the ability to execute an algebraic

specification using *term rewriting* techniques in order to observe and validate the behaviour of the ADT's operations on example data.

Chapter three reviews the existing use of computer graphics techniques in supporting programming. It starts by looking at how advances in hardware technology offers the possibility of radically different user interfaces that are intended to tap the resources of the under-valued human visual processing faculty. Particular attention is given to user interfaces for programming environments. An argument is then presented for the advantages of multiple, synchronised views of a complex object (such as a computer program or specification) based on textual *and* graphical representations. The chapter then surveys previous work in the areas of *program visualisation* and *visual programming*. The different approaches and techniques in these areas are placed in context using a new taxonomy based on the software life-cycle. The chapter concludes with a discussion of the problems, identified in this survey, of applying computer graphics techniques to programming.

Having introduced ADTs and surveyed previous applications of computer graphics to programming, chapter four brings the two areas together for the first time. By discussing in general terms the special needs of visualising ADT specifications, the chapter highlights high-level design requirements that guided the development of VISAGE. The chapter also describes how Smalltalk-80 was chosen as the implementation language, and looks at the implications this had on the architecture of VISAGE.

Having laid the groundwork for the design of VISAGE, chapter five presents a detailed description of the development of the different graphical and textual views that are used to represent the different facets of an ADT's specification. A detailed description is given of each view including its evolution, layout algorithm and implementation details where considered important. The chapter includes a description of the VISAGE automatic help and specification commenting facility.

Chapter six concludes the description of VISAGE by discussing the extension of the basic visualisation machinery developed in chapter five to facilitate the graphical construction and editing of an ADT specification. A description is given of the development of generic and specialised graphical editing facilities. The chapter concludes with a description of the PLAYPEN, an animated, graphical term rewriting environment that allows users to create example data and have them transformed by application of the equations that define the behaviour of an ADT.

An evaluation of the VISAGE system is presented in chapter seven. The chapter discusses the evaluation method including the pilot study, choice of subjects, design of the task, user questionnaires and system tutorial. This is followed by a presentation and analysis of the

results obtained by the evaluation together with a discussion of the limitations of the evaluation.

Chapter eight presents the overall conclusions reached by the research. Experimental results from the evaluation of VISAGE suggest that embedding algebraic specification within an interactive graphical environment is indeed preferred to the traditional paper-and-pencil approach by a variety of users learning to specify ADTs formally. The case for providing an extended version of VISAGE for use by more expert users is then debated. This chapter also discusses more general points raised in earlier chapters in the light of experience gained in developing VISAGE. The main body of the thesis closes with suggestions for future work including investigating the use of colour or shape, as well as the potential of programming-by-example techniques for creating specifications.

The appendices to the main thesis contain a glossary of all the abbreviations used in the thesis; a formal description of the syntax of the VISAGE (textual) specification language; a complete specification of the stack ADT that is used as a source of examples throughout the thesis; and the set of materials used in the evaluation described in chapter seven including the *VISAGE Tutorial*, the two questionnaires and the task description.

Chapter Two
Abstract Data Types and their Formal Specification

2.1 What is an Abstract Data Type?

Human beings have an irresistible urge to group objects according to characteristics that those objects share. In computing, related data values are grouped together into a *data type*, e.g. 423, 2 and –4 all belong to the data type integer. Morris (1973) refined the notion of type by relating data values according to their behaviour under the influence of a particular set of operations. For example, Integer values all behave the same way with the integer arithmetic operations, e.g. addition. Each *data type* is therefore a set of values (called the *carrier set* of the data type) and a set of operations on these values; the carrier set has an identifier called its *sort*. For example, the Boolean data type has the carrier {TRUE, FALSE} and operations AND, OR, NOT, etc. Unfortunately, the terms *sort* and *data type* are often used synonymously in the literature.

Organising values into data types is not simply a classification exercise but also a way of improving the construction of software. One of the main reasons for using types in programming is to catch *type errors* caused by applying an operation to an incorrect type of operand, for example trying to take the square root of a Boolean value. Such errors are common and if not prevented can result in programs corrupting data, often with catastrophic results.

To the hardware of a digital computer everything appears as a sequence of binary digits and consequently may be regarded as belonging to the same, single type. The instruction set of a computer can interpret these sequences in several ways, e.g. as memory addresses, executable code, integers, etc. However, there is nothing to ensure that a bit pattern is used as it was intended, e.g. an integer may be used as a memory address or even as a machine instruction.

Usually, the values being manipulated within the program will correspond to complex entities and will need to be represented as bit sequences within the computer. Successive generations of programming languages have provided higher-level data types that are automatically mapped onto the basic machine representation allowing programs to be written at a level closer to the problem domain, for example FORTRAN allows engineers to create and operate on complex numbers in their programs without needing to know how such numbers are represented in the memory of the computer. The type information

allows the compiler to select the best representation for a particular sort of data during code generation so that operations on the data are performed as efficiently as possible.

2.1.1 Data Abstraction

Brooks (1987, p.12) argues that software systems are perhaps the most complex human construct (for their size) because no two parts are alike. To construct a large piece of software requires mastery over a huge number of details with an error in any one potentially leading to disaster. Unfortunately, human beings have a very limited capacity for handling information simultaneously: Miller (1956) presents evidence that the limit is as low as about seven *chunks* at a time in our short-term memory. Although the number of chunks is limited, their *size* can vary, for example they can be increased by imposing a structure on the problem. In this way, the volume of information that can be handled is increased. *Abstraction* is a structuring technique that partitions an object's description into levels with each level suppressing irrelevant detail in the next level. To use a cosmic example: the Universe is made up from many galaxies (level 1); each galaxy has many stars (level 2); each star is made up from many atoms (level 3); each atom has many nucleons (level 4), and so. It is clearly impossible to think of the Universe at the atomic level but a layered structure provides a way of grasping the overall picture by concentrating on manageable pieces, e.g. how galaxies are related to stars.

Abstraction techniques have been used in programming languages as a way of structuring the design of software (Liskov and Guttag 1986). One of the first abstraction mechanisms in programming was the use of subroutines to extend a general-purpose programming language with specialised, abstract operations that match actions in an application. For example, in a word processing application, actions such as formatting a paragraph or deleting a word would be handled by subroutines. These *procedural abstractions* allow the designer to think using larger chunks since the irrelevant details of how the subroutine performs its operation is now hidden from the user of that subroutine. If its internal workings are also complex then decomposition into further levels of procedural abstraction can simplify its design.

Just as procedural abstraction allows new operations to be created, so *data abstraction* extends a programming language with new types of objects that occur in an application. In the word processor example, new data types for objects such as words and documents would be introduced. These data abstractions allow the designer to think about objects in larger chunks by suppressing the details of how the object is represented using the primitive data types of the programming language. When the details of a data type's representation are completely hidden from users of that type, then it is called an *Abstract Data Type* (ADT), a term first introduced in Liskov and Zilles (1974).

Unfortunately, the facility to support the creation and manipulation of new ADTs (data abstractions) is not yet standard in contemporary programming languages. This is a serious deficiency since it means that programmers have to think in terms of the underlying representation rather than the objects used in the application:

> *"...having a facility for the definition of abstract data types within a programming language increases the likelihood that the program text will accurately reflect the thought processes that lead to its construction".*
>
> Guttag (1975, p.9)

2.1.2 Encapsulation

Although Pascal base types, for example, can be combined into larger structures, the details of the representation are not hidden and can be subverted by a malicious client. These problems are avoided in programming languages that support ADTs, i.e. the language ensures that the underlying representation of the data type cannot be accessed.

By separating the representation of a type from its use, ADTs protect data values from being accessed and manipulated in ways that might infringe their integrity:

> *"[An ADT] may be viewed as a set of clothes (or a suit of armour) that protects an underlying untyped representation from arbitrary or unintended use. It provides a protective covering that hides the underlying representation and constrains the way objects may interact with other objects".*
>
> Cardelli and Wegner (1985, p. 474)

The integrity of an ADT is ensured by requiring that the only access to the underlying representation is through the interface provided by the set of operations associated with the ADT. The details of the representation are concealed and known only within the ADT.

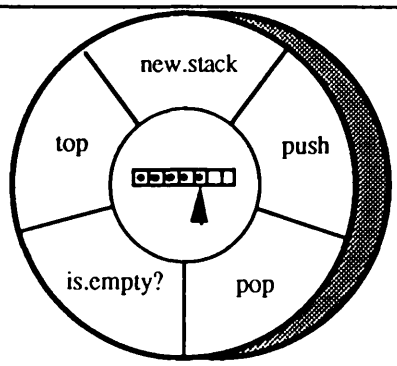


Figure 2.1: Encapsulation of a Data Structure

This desirable property is known as *encapsulation* and is supported by languages such as Ada with packages, and Smalltalk with classes. The encapsulation of a data structure (in

this case for a stack ADT) is shown diagrammatically in figure 2.1. The ADT's operations forms a protective shield around the data structure and ensure that access is only possible through one of these operations.

So long as the semantics and external interface remain unchanged, an ADT's representation may be altered, to improve efficiency for example, secure in the knowledge that it will have no effect on any programs that use this ADT. Since the details of the representation are hidden, programmers cannot exploit or make assumptions about the particular representation used: this avoids a lot of trouble if the representation is later changed.

In languages that do not support encapsulation, accesses to the underlying representation of a data type are often spread throughout the program making it very difficult to change. By restricting access to the underlying representation to the public interface operations, encapsulation also allows the *correctness* of an representation to be checked much more easily since only occurrences of the interface operations need be examined. Without encapsulation, all accesses to the representation throughout the program must be checked for correctness. Encapsulation also makes implementation and maintenance much easier since an entity is located in one fixed place. This is a characteristic feature of systems developed using Object-Oriented Programming techniques. Proponents such as Cox (1986) claim that an object-oriented approach radically improves the development of flexible, high-quality software.

2.1.3 Object-Oriented Programming

The phrase "Object-oriented programming" (OOP) is a heavily abused one having been applied to all manner of things. This thesis adopts the definition of OOP as given by Danforth and Tomlinson (1988) in their survey of the different type theories devised to explain OOP. The computational universe is composed entirely of *objects:* encapsulated data structures with an interface defined by a set of associated operations, i.e. an ADT. Computation is performed by objects sending *messages* to other objects asking for operations to be performed. This contrast with conventional approaches is captured in Ingalls' view of OOP:

> *"Instead of a bit-grinding processor raping and plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires".*
>
> Ingalls (1981)

Objects of the same form all belong to the same *class* with the behaviour of interface operations being defined by the class. Classes are arranged into an inheritance hierarchy with a class inheriting the operations and encapsulated data structures of ancestors higher

in the hierarchy, thereby encouraging the reuse of code. Danforth and Tomlinson (1988) regard OOP as the combination of ADTs and an inheritance mechanism.

2.1.4 Software Specification

Software specification takes as its starting point the requirements of the customer stating what the system should provide and any constraints on how it should do it, and converts these into a set of documents, including a contract between supplier and customer.

The development of a software system passes through several different phases in its journey from original client needs to the final delivered system. To help manage this process and to structure the sequence of phases, various software *life-cycles* have been developed that describe how the results of one phase are fed into the other phases. This thesis is concerned only with the *Software Specification* phase that corresponds to the creative task of mapping a client's requirements onto a software system. In order to understand how this phase fits in to the overall process, a simplified, linear view of software development is adopted (often called the *Waterfall* life-cycle), and shown diagrammatically in figure 2.2.

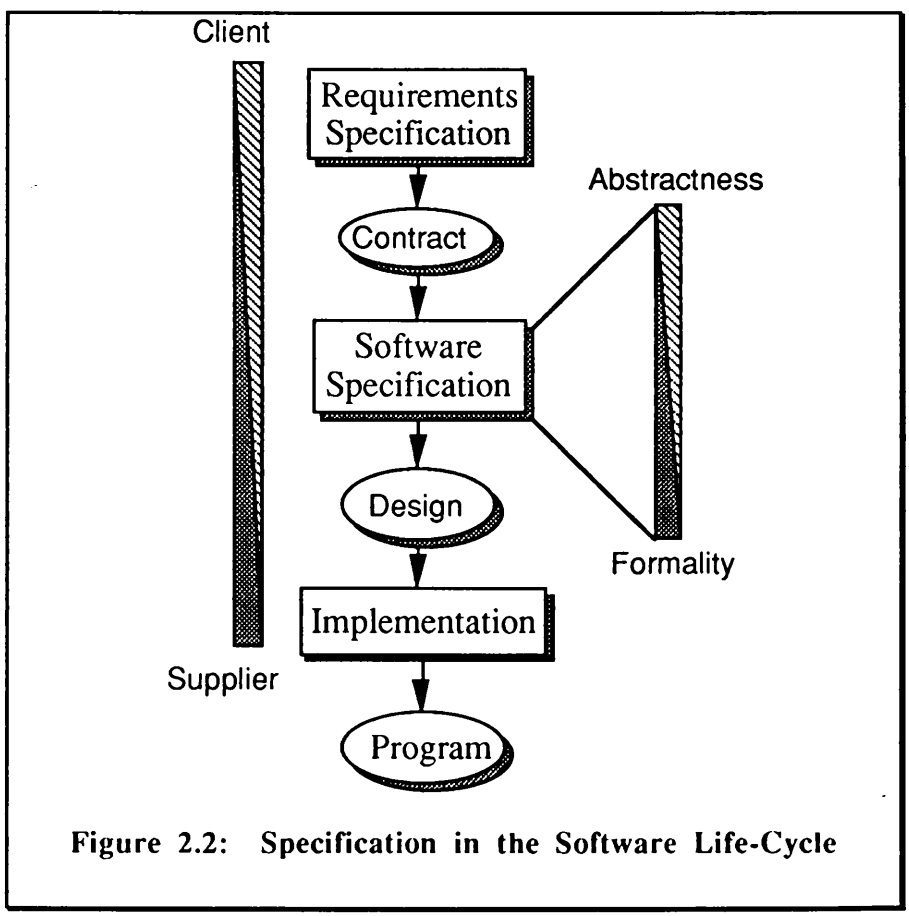


Figure 2.2: Specification in the Software Life-Cycle

In this figure, rectangles represent phases and ovals represent documents passed between phases; the sliding scales at either side of the figure indicate the relative contributions of

each attribute or agent through the life-cycle. This view does not take into account the numerous feedback loops that typically occur in system design and is therefore not an accurate temporal description of the software development process. In addition, the boundaries between phases are usually not as clear as shown in figure 2.2. A more detailed account of the overall software life-cycle, together with alternatives, is given in Sommerville (1989).

The life-cycle starts by determining the client's needs and transforming them into a requirements specification that states what services the system should provide. This document should be written with sufficient precision that it can serve as a contract between supplier and client, and also be clearly understood by both parties.

The requirements contract acts as the input to the software specification phase where the original user-centred description of how the system should behave is turned into a constructional design for the software system. In our simplified model, the specification phase can be regarded as a pipeline that refines a sequence of design abstractions from highly abstract requirements into a concrete design that can be implemented as an executable program. After the design of the overall system architecture, the design of finer details such as the system's data abstractions can be specified, as will be described in the next section.

The final phase of the life-cycle takes the complete design and converts it into executable program that hopefully meets the client's requirements. This phase requires less creative effort than for the software specification phase and considerable effort has been spent in trying to automate it using tools such as fourth-generation languages.

2.1.5 Specification of Abstract Data Types

The design of ADTs is now commonly regarded as the central problem in the detailed design of a software system, particularly in the object-oriented style. Since they act as the skeleton of a software system, it is vitally important that the ADTs are correct, i.e. exhibit the intended behaviour. The intended behaviour of an ADT (in common with other system components) is described by a *specification* which describes the values and operations of the ADT without pre-empting later implementation decisions. The specification has to be precise and sufficiently detailed to ensure against the possibility of misinterpretation, and should be understandable by all interested parties. The traditional way of specifying software is by a semi-formal description using a natural language such as English. Unfortunately, although such languages are expressive and readily understandable, they can be ambiguous, leading to misunderstanding which defeats its purpose as a way of precisely communicating a design, especially for something as pedantic as a computer:

"The whole idea of specifications is to be able to use them to check the correctness of implementations. If they are going to be used as standards in this way, one's first concern must be to ensure that they are themselves correct."

Goguen et al. (1978, p.85)

To overcome this problem, languages have been developed with a formal, mathematical basis to ensure that well-formed descriptions in that notation are unambiguous. This formal basis states just what syntactic forms are legal and gives a precise meaning to the sentences of the language. Generally such languages have been used to *program* a computer, but recently, formal languages have been developed to support the *specification* of software.

Formal Specifications

The use of formal methods to specify software is still in its infancy. Several different approaches have been investigated, each revealing particular strengths and weaknesses for certain applications: the main interest to this thesis being the formal specification of ADTs. The different approaches to formally specifying ADTs can be classified under two main headings: model-based and algebraic specifications. With the former, the operations of the ADT are explicitly specified in terms of a mathematical model, such as Set theory, which has a precise formal semantics. With algebraic specification, the operations are implicitly specified by relating them to each other using algebraic equations. A comparison and discussion of the two approaches is given in Duce and Fielding (1987) where a representative from each camp is used to specify a non-trivial example from the Graphical Kernel System.

For specifying ADTs, it seems that the algebraic approach is most suitable for several reasons. The historical development of ADTs has been based on an algebraic framework since this describes them in an elegant way and has evolved into a mature theory. The historical momentum has resulted in the algebraic style becoming the de facto standard for the specification of ADTs. The algebraic style is suitable for introducing ADTs to novices because of its simple format and syntax, and has been used in introductory texts on data types e.g. Horowitz and Sahni (1976). Model-based approaches such as Z, on the other hand, can be quite daunting with their use of obscure symbols as illustrated in the example specifications given in Hayes (1987). Providing computerised tools to support such notations is complicated by the fact that these symbols do not normally appear in standard computer character sets.

Correctness

By keeping the design of an ADT's behaviour separate from the details of the representation, the task of ensuring that they are correct becomes a two-stage process.

The first stage specifies *what* the ADT should do without regard to *how* it should do it. Since a formal specification of the ADT contains all the necessary details about how the operations of an ADT should behave, it can be verified using proof techniques before embarking on the costly implementation phase. This means that errors are discovered earlier in the development process and can be so rectified without having to undo implementation decisions. Only when the abstract specification of the ADT is correct does the second stage deal with how the ADT is represented and what algorithms are to be used to implement its operations. In traditional software development, these two stages are combined: correctness checking is therefore much more difficult as it must be done by *testing* the full implementation rather than formally *verifying* an abstract specification:

> *"Though systems have occasionally been implemented in a top-down fashion, they have for the most part been tested from the bottom up. This was necessary because the upper levels could not be easily tested in the absence of an implementation of lower levels".*
>
> Guttag et al. (1978a)

2.2 Algebraic Specification

An *algebraic specification* defines an ADT in a representation-independent manner by declaring its operations and then implicitly specifying the ADT's behaviour by stating relationships between these operations. These relationships are usually given in the form of algebraic equations†. The technique is termed *algebraic* because the ADT is regarded as an algebra: a set of values together with closed operations defined over this set.

The application of algebraic specification to ADTs was first described in Guttag (1975) and has subsequently received a large amount of attention. As described above, a formal language uses a mathematical basis to give a precise meaning to sentences written in the language. An algebraic approach is commonly used because it naturally matched the nature of ADTs by concentrating on the operations with the details of how values are represented being hidden. The necessary algebraic theory was later refined and extended, particularly by Goguen et al. (1978), into *many-sorted algebra* which has become the standard theoretical framework for describing the semantic properties of ADTs (Ehrig and Mahr, 1985). The following description of the algebraic specification technique introduces some of the basic ideas of many-sorted algebra as required; a full description of the underlying mathematics is given in Ehrig and Mahr (1985) with Cleaveland (1986) giving a good introduction to the main concepts.

---

† In this thesis, the term *algebraic* is taken to mean an equational style of specification.

2.2.1 Example Specification

An example specification of the (ubiquitous) stack data type will be developed to help introduce the algebraic specification of ADTs. A complete specification of the stack ADT is given in appendix C. Larger examples of algebraic specifications are given in Mallgren (1982) and Cohen et al. (1986), the former specifying data types commonly encountered in graphics programs, the latter specifying an electronic office.

The stack data type was chosen because it is commonly used in many programming applications, and illustrates all the main features of algebraic specification. It is also relatively simple to understand and can be specified quite concisely. To simplify the example, details of how to handle a bounded data structure are ignored and so stacks are assumed capable of holding an infinite amount of data.

Unfortunately, there are many different dialects of specification language, each developed for a single system. In this description the particular notation used is similar to the syntax used in Guttag et al. (1978b), although it can be easily converted to any of the other dialects. A complete description of the specification language grammar used in this thesis is given in appendix B.

The stack specification illustrates how a data type can be made abstract by ignoring the implementation issues. Stacks are usually explained in terms of an operational model such as an index into an array, and it is instructive to see how their behaviour can be described without recourse to such a model.

The stack ADT will have five operations associated with it as described by the following informal specification:

1. `new.stack` will create an empty stack;
2. `push` will take an existing stack and a data value and return a new stack with the data value as the top-most element;
3. `pop` will take an existing stack and remove the top-most data value, and return the modified stack;
4. `top` will return the value currently at the top of the given stack;
5. `is.empty?` will return `false` if the given stack contains any values and `true` if it is empty (`true` and `false` are declared in the Boolean ADT).

Stacks can hold different sorts of information in different applications: it is a generic data type. The common property in all these uses is first-in-last-out storage behaviour. A good specification technique will capture this essential characteristic without placing unnecessary constraints on the sort of data the stack can hold.

2.2.2 Specification Header

In specifying a large system it is natural and desirable to decompose the problem into a hierarchy of smaller independent sub-problems that are easier to understand. Eventually the problems will be sufficiently simple to specify directly, building on a library of specifications for primitive ADTs such as array, set, Boolean, etc.

Algebraic specifications are usually constructed incrementally, building on existing ADT specifications. Larger specifications can be built by *using* the sorts and operations of other specifications. For example, an integer operation to test whether two numbers are equal returns a Boolean result and therefore the integer specification needs to use the Boolean one. A specification that uses other specifications can itself be used, with the result that the specification of a complex system often forms an acyclic graph of usage dependencies.

The new ADT is likely to use operations defined in existing ADTs, and these have to be declared before they can be used. With the stack example, `true` and `false` from the Boolean ADT are used. The generic stack header is therefore declared as:

```
Datatype:  stack[data] uses: Boolean;
"A generic first-in-last-out storage structure."
```

where `data` is the formal parameter for the generic data type and can be instantiated with any actual type, e.g. `stack[integer]`. For ADTs that do not require a generic argument, e.g. stack-of-integers, the generic part of the header is omitted.

The stack header states that three sorts of entity are used in the specification: stack, data, and Boolean each having a corresponding carrier set in the many-sorted algebra, e.g. the sort Boolean refers to the set of values `{true, false}`. The `Datatype` declaration implicitly introduces a single new sort which is the same as the ADT's name. In the above example header, the new sort stack is being introduced. More sophisticated languages, such as OBJ (Futatsugi et al. 1985), explicitly introduce one or more new sorts.

Allowing specifications to be parameterised makes them attractive as a unit of software reuse. In a practical setting, the parameter mechanism can be generalised to allow constraints to be imposed about what sort of type can appear as an actual argument, e.g. the specification could insist that the argument type must have an equality operation. Examples of such parameter restrictions are given in Cohen et al. (1986), and although very useful in a development environment to ensure proper use, they are not given further consideration in this thesis. A theoretical treatment of parameterised specifications is given in Thatcher et al. (1982).

The quotation marks in the header delimit an informal comment about the ADT. It is considered good practice to add informal comments to complement the formal specification

so as to explain what the ADT represents, what the operations do, and how they behave. Studies such as those described in Sheil (1981) have shown that comments make programs easier to understand and modify. The use of comments in something as abstract as a specification can confidently be expected to yield even greater benefits.

2.2.3 Syntax

The syntactic part of the specification (called the *signature*) introduces the operations associated with the ADT. Each operation is declared by giving its name (the *operator)* and the sorts of any arguments and the sort of the result. To illustrate this, consider the signature below for the stack ADT:

```
new.stack:                  → stack
push:          data × stack → stack
pop:           stack        → stack
top:           stack        → data
is.empty?:     stack        → boolean
```

On each line, the identifier before the colon is the operator. The component following the colon is known as the *arity* of the operator and has two parts. The list of sort names between the colon and the arrow give the *domain* of the operation: the `new.stack` operation takes no arguments and is effectively a *constant* of type stack as it always returns the same value†; `push` requires two arguments, the first of sort data (the generic values to be stored on the stack) and the second of sort stack (the × symbol is the tuple constructor), and so on. The mandatory sort name after the arrow specifies the *range* of the operation, in the above example, the `push` operation has a range of sort stack, whilst `top` has a range of sort data.

The signature of a specification is often shown using a graphical notation known as a *Signature Diagram* to indicate the sorts and operators. The signature diagram for the stack ADT is shown in figure 2.3.

In the diagram, carrier sets are shown as ovals; operations are shown as many-tailed arrows with the head of the arrow (shown emboldened) indicating the result of the operation and the tails indicating the argument sorts. The dot where the head and tail of the arrow meet is labelled with the operator. The diagram does not give an indication of the order of the operands. Such diagrams appear in many descriptions of ADTs and their specifications, often complementing the mathematical definition in works such as Goguen et al. (1978).

---

† Nullary functions that return different values, such as random number generators, cannot be handled within this framework.

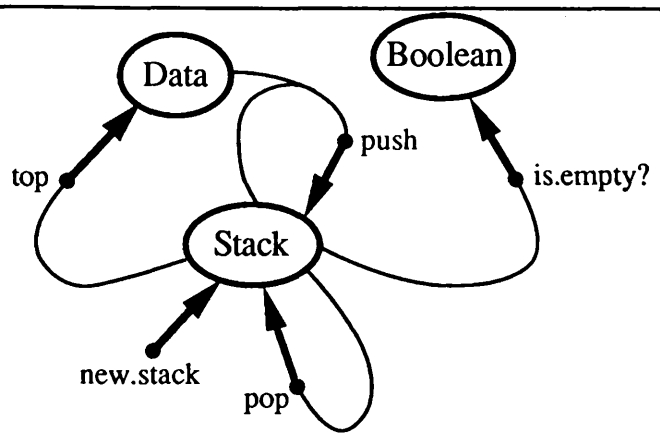


Figure 2.3: Traditional Signature Diagram for the stack ADT

Operation Classification

Following the approach of Liskov and Guttag (1986), operations can be partitioned into two main groups depending upon the sort of result produced:

- *Observer* operations return a value whose sort is *not* the same as the sort introduced by the parent ADT. They are used to return the attributes of an object. In the stack example, the observer operations are `is.empty?` and `top` since they return results of sorts Boolean and data respectively;
- *Generator* operations return a value whose sort is the *same* as the sort of the parent ADT. In the stack example, the generator operations are `push`, `new.stack` and `pop` since they all return results of sort stack. Every value of a particular sort can be built up by applications of a small, finite subset of generators called *constructor* operations belonging to the corresponding ADT. In the stack example, the constructors are `new.stack` and `push`. The `pop` operation is not regarded as a constructor because any stack built using `pop` can always be replaced by one using only `push` and `new.stack`.

Note that there must be at least one *constant* constructor operation to act as the starting point for all other values, as otherwise the carrier set will be empty. For example, we need a `new.stack` before data items can be pushed on.

Terms

The signature of an ADT is simply a context-free grammar for constructing legal expressions involving the operators declared or imported into the ADT's specification.

This can be demonstrated by (partially) converting the above declarations into the equivalent Backus–Naur Format (BNF) production (with symbols in upper case representing non-terminals):

```
STACK  ::=  new.stack | push (DATA, STACK) | pop (STACK);
```

A well-formed expression defined by the grammar is a string of operator symbols known as a *(ground) term*, with the expansion of the non-terminal symbols ensuring that the term is type-consistent. Terms have a functional appearance very similar to that of LISP. The set of all terms that can be generated by a signature is called the *Word Algebra* of the signature with each individual term being composed of a finite number of operators. Given a signature, automated tools such as type-checkers and syntax-directed editors can be used to create valid terms: an example of a specification environment that includes such facilities is ASSPEGIQUE, described in Bidoit and Choppy (1985).

Examples of terms of sort stack (where `zero` and `one` are constant terms of sort integer) include:

1. `"new.stack"`
2. `"push(top(push(zero,new.stack)),pop(push(one,new.stack)))"`

Usually, the quotation marks around the string are dropped for convenience. Note that all the operators are regarded as prefix symbols; a more liberal grammar (such as that used by the OBJ2 parser described in Futatsugi et al., 1985) would allow *mixfix* syntax where arguments can be placed in an arbitrary position relative to the operator, e.g. `3+4` instead of `add(3,4)` where + is the operator for the addition operation. Although this improves readability, such conveniences are not used here as they introduce unnecessary complexity into the grammar.

A convenient way to think of terms given their syntactic nature is as a parse tree. This allows the structure of the term to be easily grasped by removing the clutter of the concrete syntax. An example of such a tree is given in figure 2.4 for the term: `push(top(push(zero,new.stack)),pop(push(one,new.stack)))` shown above.

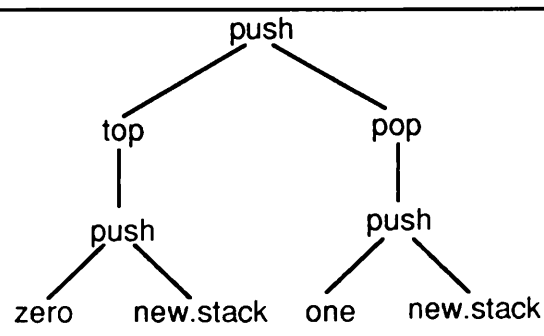


Figure 2.4: Example of a Term's Parse Tree

The tree diagram reflects the recursive structure of the term with sub-trees representing complete sub-terms. The leaves of the tree represent constant operations and variables with the internal nodes representing operation application.

Terms with Variables

In mathematics the concept of a *variable* allows a single statement to made about a possibly infinite number of situations. The same mechanism is useful in algebraic specifications to let a term involving one or more variables (an *open* term) stand for a possibly infinite number of ground terms by having each variable replace sub-terms of the appropriate sort. The equivalent set of ground terms can be obtained by substituting all possible terms of the appropriate sort for these variables. An example of an open term of sort stack is `pop(push(d,s))` where `s` and `d` are variables of sort stack and data respectively.

2.2.4 Semantics

The third part of an algebraic specification describes the semantics of the ADT by showing which terms in the word algebra should have the same value in the algebra that is acting as a model of this ADT. This is done by specifying a *congruence relation*† on the word algebra using a set of equations (an *equation* being a pair of terms, written as `s = t`, where `s` and `t` are terms). This relation partitions the word algebra into congruence classes such that all the members of a class map onto the same value in the algebra, i.e. the terms are *equivalent.*

A congruence relation is said to *satisfy* a set of equations if, for every ground equation `s = t`, the terms `s` and `t` are in the same congruence class. In general there may be many congruence classes that satisfy a set of equations necessitating a choice as to which one will be used when representing the ADT. The most common approach (and the one adopted in this thesis) is called *Initial Semantics* and assumes that the congruence used is the reflexive, transitive and symmetric closure of the equations (with the confluence property). This regards terms as belonging to different congruence classes (i.e. represent different values) unless they can be proven equivalent by the equations of the specification. A discussion of alternative approaches is given in Bauer and Wössner (1982, pp. 195–204); a good introduction to this topic is given in Cleaveland (1986, pp. 189–202).

---

† A congruence relation R on a set is an equivalence relation (i.e. it is reflexive, symmetric and transitive) with the additional property that if $R(x_i, y_i)$ for $1 \leq i \leq n$, then $R(h(x_i), h(y_i))$ where h is some operator from the signature.

Examples of Equations

Consider the two equations below that describe how the `is.empty?` operation behaves. The `is.empty?` operation takes a single argument of sort stack and returns a value of sort Boolean:

```
For all d:data, s:stack
     is.empty? (new.stack)     = true          Eqn. 1
     is.empty? (push (d, s))   = false         Eqn. 2
```

The *For All* statement binds terms of a particular sort to the variables `d` and `s` and is regarded as applying to both equations (although the first equation has no variables). Equation one asserts that a newly created stack is empty, which we will intuitively assume to mean that it contains no data values. Since a term is used to represent a value of a particular sort with each value being represented by possibly many different terms, an alternative view of this equation is that the term `is.empty?(new.stack)` represents the same value as the term `true`.

Equation two states that *any* stack created by pushing a data value `d` onto a stack `s` will not be empty, where the arguments `d` and `s` are variables of sort data and stack respectively.

The two equations above state a property of stacks without adding the irrelevant representational details to the specification. A description in terms of a machine model would need to define what it means for a stack to be empty, for example, in terms of the index into an array. This is overly detailed and biases the implementation of the stack ADT, so that when the ADT's representation is later considered, alternative yet equally valid implementations are perhaps excluded from consideration.

The examples of equations given so far have been quite simple having right-hand terms consisting of either variables or constant terms. In general, equations will be more complicated with the right-hand terms of arbitrary complexity often involving recursion equations. Consider, for example, the two equations below that define the behaviour of an operation called `size?` that returns how many items have been pushed onto a stack:

```
For all d:data, s:stack
     size? (new.stack)     = 0                     Eqn. 3
     size? (push (d, s))   = 1 + size? (s)         Eqn. 4
```

Equation three states that a newly-created stack contains no items and establishes a base case for the recursive definition. Equation four states that the number of items on a stack after pushing on a new item is one more than the number on the original stack. Note that the equation is recursive since the `size?` operator appears on both sides. Although it may not be obvious that this fully defines the `size?` operation's behaviour, a simple inductive

proof can show that indeed it does. An example will show how the operation works (where the data items are text characters):

```
size? (push ('k', push ('e', push ('v', new.stack))))
      = 1 + size? (push ('e', push ('v', new.stack)))     by Eqn 4
      = 1 + 1 + size? (push ('v', new.stack))             by Eqn 4
      = 1 + 1 + 1 + size? (new.stack)                     by Eqn 4
      = 1 + 1 + 1 + 0                                     by Eqn 3
      = 3                                                 by arithmetic
```

Conditional Equations

Occasionally an equation is needed that says a term is equal to one of two other terms depending upon the value of some condition. Consider the following definition of an operation that given a data value and a stack will return the number of times that data value appears on the stack. (The operation `equal?` is assumed to return a Boolean value and `succ` returns a number one larger than its argument).

```
1.  count (d, new.stack)     = 0
2.  count (d, push (e, s))   = IF equal? (d, e)
                                  THEN succ (count (d, s))
                                  ELSE count (d, s)
```

In the second equation, the right-hand term is conditional on the result of the term `equal?(d,e)`, being `succ(count(d,s))` if the result is `true` and `count(d,s)` if `false`. The ELSE term is optional and can also have nested conditionals. The conditional mechanism can be regarded as a pre-defined operator in every sort, e.g. for integer ADT as used above:

```
if-then-else: boolean × integer × integer → integer
```

2.2.5 Choosing Equations

Guttag et al. (1978b) give some rules-of-thumb for deciding just what equations are needed in a specification. The first step is to decide which operations in the ADT are to be regarded as constructors i.e. the set of operations that can represent all values of the type. The second step is to write equations that show how all the other operations behave on values of the data type built from each of the constructors. In the stack example there are two constructors: `push` and `new.stack` and three other operations which means six equations in all will be required:

```
1. is.empty?(new.stack)=...          2. is.empty?(push(d,s))=...

3. top(new.stack)=...                4. top(push(d,s))=...

5. pop(new.stack)=...                6. pop(push(d,s))=...
```

Although these rules-of-thumb are a useful guide, some specifications may require more equations in order to specify a special property of an operation, e.g. the following equation may be added to the integer specification to handle associativity of the addition operation:

```
add(a,add(b,c)) = add(add(a,b),c)
```

2.2.6 Problems with Algebraic Specification

The definition of an operation as used above was a function free from side-effects with each operation returning exactly one result. This may be unnatural for operations such as `pop` in the stack ADT which is often thought of as returning the last pushed value *and* removing it from the stack as a side effect. Having to specify everything in terms of functions may cause trouble during implementation since multiple copies of large data structures may be created. However, it must be remembered that the specification is only trying to get the ideas right with no regard for the implementation. Although a purely functional approach is used to specify an ADT, there is no requirement to implement it that way.

As an example consider the specification of a file system ADT with a typical operation to add a new file to the file system. This would typically be specified as an operation that takes the new file and the existing file system and returns a *new* file system which implies having two copies of the file system: before and after calling the add operation. Clearly, a practical implementation would simply update the file system rather than create a new one with the extra file. Instead of the add operation being a function, it is perhaps better specified as a procedure where the file system is passed using a call-by-name parameter mechanism. Guttag et al. (1978b) proposes an enhancement the standard algebraic technique to allow call-by-name parameter passing as a way of handling side-effects such as in the add operation described above.

Although the recursive nature of the equations in an algebraic specification is elegant, novices find it very difficult to understand or even believe that it works at all! Even more experienced users may find it difficult to visualise how a recursively defined operation works and will work through examples to help understand the behaviour of the operation. A practical specification environment has to support this by letting users experiment with examples that illustrate the recursion.

Validation and Verification

A software design is validated by confirming that it matches the customer's requirements. If a formal specification of system is to act as a contract between customer and supplier, then it is essential that the customer understand it: this is unlikely if it is expressed in obscure mathematical notation. A design is verified by checking it against its specification. Programmers without training in formal methods find verifying a design against a formal specification to be quite daunting. As Cohen et al. (1986, p.107) note: *"Probably the most successful [formal] methods to date have been where the general approach is, to some extent, familiar to software engineers"*. These problems can be overcome in several ways, for example: by increased training in formal methods; or by developing tools to provide a more palatable front-end to a formal specification environment.

Error Handling

During the execution of a program an error may arise in using ADTs in situations outwith their normal operating range, e.g. trying to access the top-most value of a newly-created stack. It is important that such errors are handled properly, otherwise the system may behave in unexpected ways. Usually an error message is produced that explains what went wrong. The specification of the system should explicitly describe what situations cause errors and how they should be handled. Unfortunately, the formal specification of errors in ADTs introduces subtle theoretical difficulties such as stating the type of the error values, and what operations should do if given an error value as an argument. For example, consider the division operation for natural numbers with the signature:

```
nat × nat  →  nat
```

This signature states that the result of the `div` operation will be of sort nat. If this is so, then the undefined value of `div(1,0)` must be of sort nat. To overcome this problem, the value `error` could be added to sort nat. However, this just makes things worse since we now have to consider how to handle terms such as `div(error,0)` since the signature states that `div` is defined over any value of sort nat. An alternative approach is to change the signature of `div` to use a *subtype* of nat that does not include zero (i.e. the carrier set is a subset of nat's carrier set). The expression `div(1,0)` would result in a type error since zero is not included in sort non-zero-nat.

```
nat × non-zero-nat  →  nat
```

Further examples of these and other problems are given in Goguen et al. (1978) together with pointers to their solution. A theoretical treatment of error handling is outwith the scope of this thesis although a simplistic, practical mechanism is discussed in chapter five.

2.3 Term Rewriting

The simplified Waterfall model of software development discussed in section 2.1.4 delays implementation until the software specification phase has produced a complete, detailed design. However, the formal specifications of ADTs developed towards the later stages of specification offer the chance to leap-frog the traditional implementation phase by allowing these formal specifications to be directly executed† given a suitable interpreter that obeys the semantics of the specification language. This concept of an *executable specification* blurs the distinction between a specification and a program: formal specification can now be regarded as programming in a very high level language. Executable specifications are particularly useful in checking that operations behave correctly before starting on the expensive process of implementing them in a programming language. This has the benefit that a client can be shown a working prototype system at an early stage in the development process to validate against the requirements. Although the prototype will not execute with the efficiency of the final implementation the client will be able to say what needs to be changed before the costly implementation phase.

Algebraic specifications can be executed using a technique called *term rewriting*. An equation in an algebraic specification asserts that two terms represent the same value, i.e. from a semantic viewpoint they are synonymous: every occurrence of one term can be replaced by the other with no loss of meaning. The rewriting mechanism is specified by a set of rewrite rules that say how one term should be replaced by another, usually simpler, term with the same value. Rewrite rules are written as S $\Rightarrow$ T where S and T are both terms, with T being "simpler" in some way than S. Since a rewrite rule is simply an ordered pair of terms, the equations that make up an ADT specification can be regarded as rewrite rules by placing an ordering on them: usually left to right. Therefore, the equation `is.empty?(new.stack) = true` may be converted into the corresponding rewrite rule: `is.empty?(new.stack)` $\Rightarrow$ `true`. Term rewriting is receiving considerable attention from both theoretical and practical viewpoints. It has even been advocated as a general-purpose computing technique:

> *"Equational programs can serve as a useful programming language in a diversity of applications. The great strength of this approach lies in the simplicity of the semantics, which is accessible even to the non-specialist."*
>
> Hoffman and O'Donnell (1982, p.108)

---

† The term *animation* is often used in the literature to mean the execution of a specification. This thesis will use the term *execution* to avoid confusion with graphical animation.

O'Donnell (1985) gives a thorough discussion of term rewriting as a practical computing technique, including the theoretical basis, problems and future research directions in this area. The rest of this section gives an introductory overview of term rewriting and its relationship to ADTs.

2.3.1 An Example of Term Rewriting

Consider the following rewrite rules from Boolean algebra (with `not`, `or` and `and` representing logical negation, disjunction and conjunction respectively):

```
1.  not(not(P))      ⇒ P
2.  or(not(P), P)    ⇒ True
3.  not(and(P, Q))   ⇒ or(not(P), not(Q))
4.  and(not(P), P)   ⇒ False
5.  not(False)       ⇒ True
```

Using these equations, the term `not(and(A,not(A)))` can be rewritten as follows (figure 2.5 shows a graphical representation of how the parse tree of the term is changed by the rewriting process):

```
not(and(A,not(A)))  ⇒ or(not(A),not(not(A)))   (using equation 3)
                    ⇒ or(not(A),A)             (using equation 1)
                    ⇒ True                     (using equation 2)
```

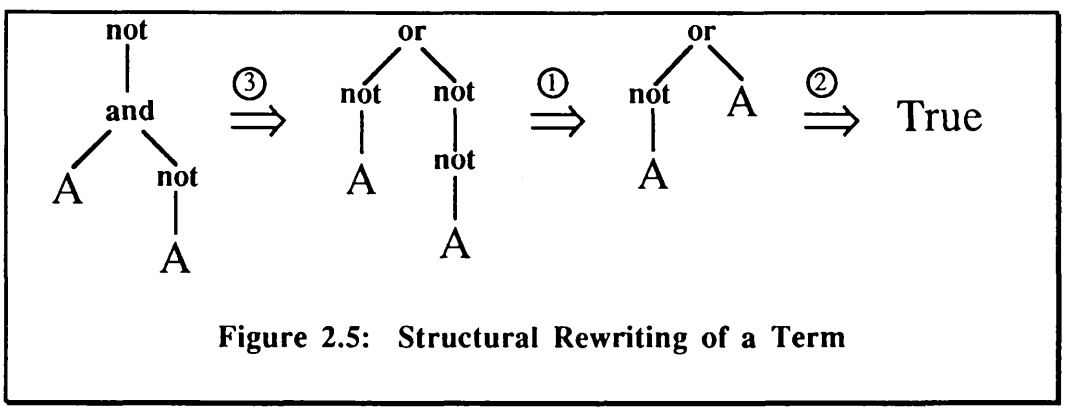


Figure 2.5: Structural Rewriting of a Term

The final term is regarded as the *answer* as it cannot be further simplified with the current set of rewrite rules and is said to be in *(canonical) normal form.*

2.3.2 Termination and Confluence of Rewriting

When equations are used as rewrite rules special properties have to be considered. The first of these is *termination* which guarantees that the application of rewrite rules to terms will not go on forever: clearly, very desirable in practical computing. A set of rules is guaranteed to terminate if they always produce a sequence of terms getting smaller in size, according to some numerical measure. For example, in figure 2.5 the sum of the

distances from the root of a tree to each of its leaves is always less than or equal to that of the previous tree: 5, 5, 3, 0.

Rewriting rules follow the declarative style of programming by not specifying the order in which rules should be applied. It is often the case that more than one rule can be applied to a term at any one time, and so it is important that the same normal form answer is produced regardless of the order in which rules are applied. For example, the term `not(and(A,not(A)))` can be rewritten in a different way to that shown above:

```
not(and(A,not(A)))  ⇒  not(False)      (using equation 4)
                    ⇒  True            (using equation 5)
```

This property is called *local confluence* (Bundy 1981, p.110) and is shown diagrammatically in figure 2.6, where the arrows between terms represent rewriting and are labelled with the rule that applies at that point. Figure 2.6 shows that the start term can be rewritten using either of two rewriting sequences into the canonical term, `True`. Confluence guarantees that so long as the rewrite rules are terminating, the same final term will be produced no matter what route is taken from the start term.

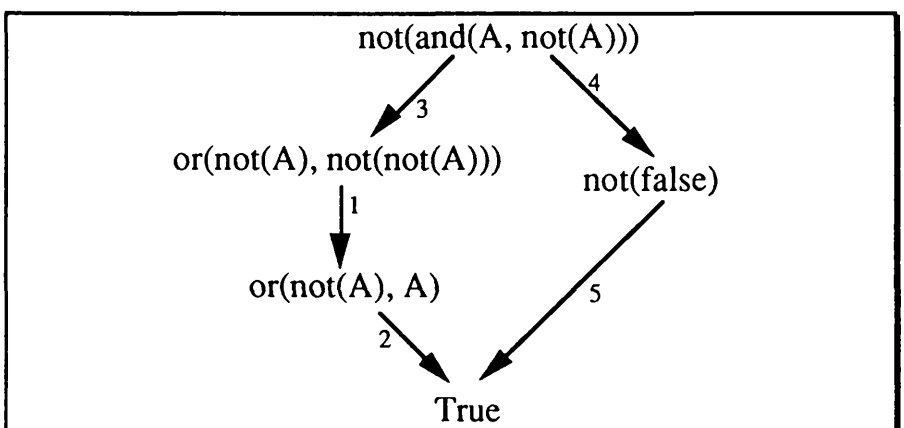


Figure 2.6: Local Confluence of Term Rewriting

Assuming a canonical form exists, given a choice of routes from the start term to a canonical form, we naturally want to take the shortest route so as to minimise the amount of work and time required for the computation. In addition, some routes may involve rewriting sequences that do not terminate and these need to be avoided. Various strategies for applying rewrite rules have been developed that impose some control over rule application in an attempt to find the optimal route to the canonical form; the main strategies are *inside-out* and *outside-in* (or normal-order) application.

Inside-out application works from the bottom-up by rewriting the argument terms before getting to the outermost operator. Outside-in application tries to rewrite a term at the outermost levels and only rewrites inner terms if no rules apply at the outer level. To

illustrate the difference between these two strategies, consider the following rewrite rule for an operation that simply returns the first of its two argument: `first(a,b)=a`, and what happens in trying to reduce the term `first(1,1/0)` to its canonical form.

With inside-out application the two argument sub-terms would first be rewritten. Unfortunately, the second term will require an infinite rewriting sequence†. Outside-in avoids this by immediately invoking the above rule to rewrite the term to the canonical form `1`. The second argument term was not even considered in this latter strategy. Since it can often avoid a non-terminating rewriting route and will only perform rewrites that will have to be performed at some point in the rewriting route, outside-in application is regarded as the superior evaluation strategy, Bundy (1981).

Term rewriting is a useful technique for showing that two ground terms are equivalent by showing that they can be rewritten to the same normal form using rewrite rules based on the equations in the specification. A theoretical treatment of this mechanism is given in Ehrig and Mahr (1985).

2.3.3 Applications of Term Rewriting

Several systems have been developed to explore the use of term rewriting as a way of computing at a very high level of data abstraction. Examples include the OBJ family of systems developed at SRI International to investigate the equational specification of ADTs with particular emphasis on error handling. The OBJ2 member of the family is described in Futatsugi et al. (1985). OBJ allows specifications to be parameterised and decomposed into modules to simplify the design process and encourage reuse of design components. The distinction between specification and programming becomes somewhat blurred since specification is now just programming in an Ultra High Level Language (UHLL) with specifications being executable within an interactive environment. Although this facility is useful for checking the design of the ADT by watching its run-time behaviour, the interpreted nature of the language means that execution is slow, making the language only suitable for prototypes. The OBJ group are trying to overcome this problem by building a dedicated *rewriting engine* to execute rewrite rules very efficiently (Goguen, 1986).

2.4 Summary

Abstract data types (ADTs) are regarded as an important development in the construction of software. By encapsulating the underlying representation and limiting access to a small number of trusted operations, ADTs allow changes to the representation to be made secure

---

† For example, if division is specified as repeated subtraction.

in the knowledge that users of that ADT will not be affected, so long as its semantic behaviour is preserved. This property makes software easier to maintain.

The behaviour and form of an ADT is precisely described using a specification that hides the irrelevant details of how the values of the ADT will be represented and how the operations that work on those values will be implemented. To ensure that the behaviour of the ADT is given a precise specification, formal specification languages with a sound mathematical basis have been developed. This chapter contains a description of the most common approach based on abstract algebra, illustrated by an example specification for the stack ADT.

An algebraic specification of an ADT can be investigated using a computational technique called term rewriting whereby the equations used to define the behaviour of the ADT's operations are used to evaluate expressions involving the operators of the ADT. This mechanism allows prototype implementations to be created very quickly to validate the ADT's behaviour against the customer's requirements.

Chapter Three
Graphical Support for Programming

"Visual programming is generally thought of as the marriage of software production methods to the visual power of computer graphics. The goal is to create a programming methodology that is considerably better than the sum of its two parts."

Grafton (1986)

3.1 Enabling Technology

User interfaces to computers were until recently strictly text-oriented, restricted to using a limited alphabet of letters, digits, punctuation and special characters. Such interfaces tend to be static and unnatural, displaying information as if it were on a long continuous scroll being viewed through a narrow aperture, merely imitating the medium of paper whose form and conventions have changed little since the Middle Ages.

Powerful workstations with bit-mapped displays allow high quality, graphical images to be combined with text to create a totally new kind of user interface, with the chance to break away from the rigid conventions of the past. Workstations usually have a pointing device, such as a mouse, enabling a user to interact with the display in a much richer way than is possible in a text-oriented system. Workstations are designed primarily as single-user machines allowing the full resources of its computer to be used in more demanding applications. Plentiful memory and processing power mean that improved interfaces can be built using, for example, colour and dynamically changing displays.

3.2 Is a Picture Worth a Thousand Words?

Human beings are visually-oriented creatures with a brain specialised over millions of years to processing information received by the eyes. A two-dimensional image on the retina is processed almost instantaneously by the brain. A picture can therefore be assimilated very quickly, making it a good way to communicate a concept, for example, in road signs where a graphic image gives warning of a danger ahead in a quick glance.

Archaeological evidence, such as the cave paintings at Lascaux in France, show that pictorial communication was being used by early man, being later refined into complex languages such Egyptian hieroglyphics. However, such languages suffer from several obvious limitations such as lack of compactness and generality, and the need for skill in

drawing. Textual languages gradually evolved over the last few thousand years as a way of overcoming the limitations of these pictorial languages, and are now the standard way of recording information and for communicating ideas between people. This was because *"it was more difficult to generate and manipulate visual material than to generate linguistic material of comparable richness and complexity"*, Davis and Anderson (1979, p.123). The past tense is used here, because computer technology now offers the possibility of generating and manipulating visual material almost as simply as textual material.

Textual language introduces a level of indirection by using abstract symbols to represent a concept rather than a evocative pictorial symbol (*icon*), for example *apple* instead of . Considerable mental translation is required before the abstract symbols can be understood in terms of the underlying concepts. A similar translation is required in converting the mental representation of a concept into a textual form. These translations, reading and writing, take considerable time to perform when compared to processing pictorial information, and are skills that require about a decade of active learning to master. Textual languages can, however, describe in a compact notation, concepts of great richness and complexity. If these languages are given a formal semantics then these concepts can also be described with absolute precision. Their generality, compactness and precision make them attractive for programming a computer.

Smith (1977) argues that a person's problem-solving abilities are hindered by the need to translate between the mental and textual representations of a solution. This is based on the evidence that mental representations are rarely linguistic. For example, in a survey of prominent mathematicians, Hadamard reports:

> *"Practically all of them...avoid not only the use of mental words but also...the mental use of algebraic or any other precise signs. The mental pictures of the mathematicians whose answers I have received are most frequently visual"*.
>
> Hadamard (1945, p.84)

Smith argues that a computer with a graphical interface would allow the use of representations nearer those used in the mind, reducing the amount of translation required to map a solution onto its representation on a computer, and thereby increasing the power of the computer as a problem-solving tool. The situation is shown diagrammatically in figure 3.1.

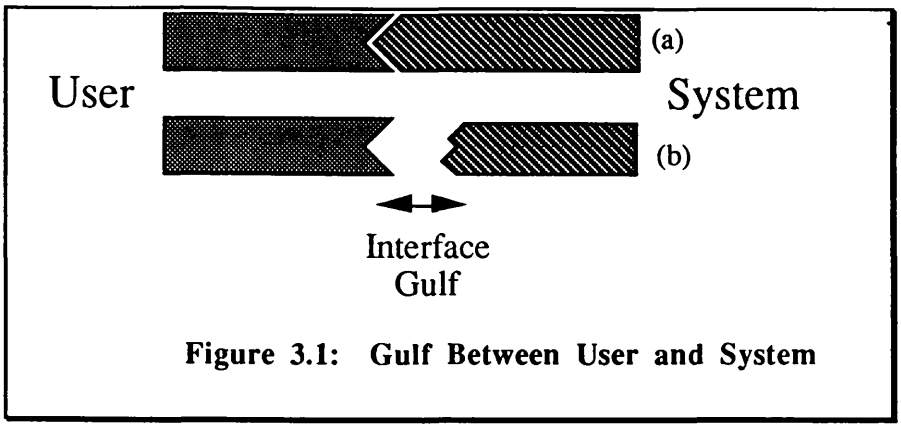


Figure 3.1: Gulf Between User and System

In figure 3.1(a) the system and user are separated by a clean, dove-tailed interface with a small translation distance; this makes it conducive to easy and efficient communication and hence problem-solving. The system in figure (b), on the other hand, presents a ragged interface requiring considerable effort on the user's part to bridge the gulf. Clearly, it should be the goal of every interface designer to build a system that presents an interface that is closer to figure (a) than (b).

3.2.1 What Are Pictures Good For?

Graphical images, by their nature, can show a large number of objects simultaneously. By exploiting a vocabulary of constructs such as shape, colour, size, texture and connectivity, a multi-dimensional space can be represented by a two- or three-dimensional picture. The ability of the brain to identify quickly the patterns in visual images means that relationships between objects encoded graphically can be spotted faster than with text, as illustrated by comparing a histogram plot to a list of numbers.

A picture can also give concrete form to an abstract entity allowing it to be manipulated as it were physical. Raeder (1985) claims that this not only helps a specialist to form and communicate ideas faster, it give novices a handle on unfamiliar material. This property has been exploited in "desktop" interfaces to computers such as the Xerox STAR described in Smith et al. (1983), where small pictures called *icons* represent abstract entities such as files, directories and electronic mailboxes, as well as physical devices such as printers. Giving an entity a physical existence also allows it to be referred to simply by pointing, without needing to be given a name as required in traditional programming languages and operating systems. Linton and Powell (1983, p.17) claim that:

> *"The ability to name by pointing adds significant power to the programming environment".*

Despite the naturalness and apparent power of pictures, the visual image has been gradually replaced by text as the dominant medium of communication, especially in formal areas such as mathematics. Even intrinsically visual subjects such as geometry have been

given textual "respectability" within algebra. Davis (1974) identifies several reasons for this, e.g. the possibility of the eye being deceived due to optical illusions.

Nevertheless, manually-drawn pictures are still used informally to help understand and reason about complex objects. For example, when programmers are trying to understand how an algorithm works they usually draw diagrams of the data structures involved to follow what is going on. Graphics are better for representing complex data structures as they naturally show the relationships between the different parts of a structure. As Duisberg notes:

> *"Data structures are almost invariably represented by two-dimensional diagrams in texts and documentation with the clear implication that the concepts and structures are more easily comprehended and understood in this spatial, visual form, than, say as a textual description or algebraic specification".*
>
> Duisberg (1986, p.131)

Graphics can also convey additional information that text cannot. For example, properties such as type compatibility can be shown by two objects having the same shape. Other graphical properties such as colour, size, distance etc. can be used to represent esoteric entities such as exception conditions, break-points and dependencies all within one picture.

A graphical representation is better than text for editing the higher-level structure of a program as it allows the structures to be manipulated directly rather than textually editing their representation as sequences of characters. Errors due to ambiguities in the textual form are eliminated, for example, the famous *dangling else* problem can be avoided if the programmer directly manipulates a graphical representation of the parse tree.

3.2.2 Synergy of Text and Graphics

The recent ability to animate and manipulate graphical images on a computer has been seen as a way of solving many of the problems in writing software, by replacing textual languages. This is dangerously optimistic. A quick glance through almost any textbook would show that graphical and textual information happily co-exist, with each complementing the other. For example, in a computing textbook, the text supplies the background and theoretical principles with graphical images being used to illustrate and clarify the text. The combination of text and graphics is more powerful than each alone, and it seems sensible to follow this approach in designing programming systems; this was recognised in the early work in this area:

"The development of a graphical programming capability must not be considered in the light of replacing written programming. Rather, we must extend the options open to the programmer so that he may use whichever method is most convenient for a particular application."

Sutherland (1966, p.18)

The new graphical technologies allow the use of many different representations, both textual and graphical, at the same time, to show the different aspects of a program in a way likely to give the greatest insight to the programmer. Perhaps the greatest hurdle to the everyday use of graphics in the programming process is an educational one. Graphics is somehow regarded as best for beginners while text is best for experts. It is commonly felt that although pictures may be useful for teaching novices, they do not have the precision and descriptive capability required for the exacting task of describing a "real" program.

3.2.3 The Need for Examples

A common technique in trying to understand how an algorithm works is to try it out using some example data. By operating on a concrete example, the behaviour of the algorithm is made visible and easier to follow. In observing the way one or more examples are transformed, the user can form a mental model of the way the algorithm works.

Many programming systems have exploited this property in different ways. Much research has been done in Artificial Intelligence to use inference techniques to create programs given examples of their input and output taken from simple domains such as list processing. Myers (1989) terms such systems *Programming-by-Example*†, but since they do not necessarily use graphics, they are not considered in this thesis.

The PYGMALION system described in Smith (1977) is an early example of what Myers terms a graphical *programming-with-example* system. The system creates a program by "watching" the user graphically manipulate example data. The system automatically converts this manipulation of a specific example into a procedure that may be applied to other data of that form. Since the results of an editing operation are immediately visible, the user sees instantly whenever a mistake has been made and can correct it straight away by editing it to the desired form, with the result that programs are often written correctly first time. This style of programming places emphasis on *showing* the computer what to do rather than *telling* it, as with traditional programming languages.

---

† Many of the terms used to describe visually-oriented systems for programming have been used in different ways by different authors. This thesis uses the terminology of Myers (1989).

The programming-with-example technique has been heavily used in visually-oriented systems because abstract objects can be given a concrete form through a suitable graphical representation. These objects can then be directly manipulated using graphical actions such as pointing and moving. An equivalent textual approach would require objects to be named before they can be manipulated, which interferes with the direct nature of the approach.

3.2.4 Multiple Views

Complex objects in the real world invariably have more than one facet each of which shows a separate aspect of its structure or behaviour. If that object is something physical, such as aircraft, then an effective representation can be produced by trying to mimic the behaviour and structure of the object in the computer. A successful example of this approach is an aircraft simulator for training pilots that recreates the layout and handling characteristics of the real aircraft. Such simulators are so realistic that pilots can qualify without ever taking to the air in a real aircraft.

However, if the object to be visualised is an abstract entity without physical form then the representational problem is more difficult. As Brooks (1988, p.8) advises:

> *"The best visualisation strategy for abstractions is ... quite different than for [physical] objects. Rather than working on making one visualisation ever-closer to the ideal, the view-maker should devote his energies to the production of many different visualisations."*.

Visualising an abstract object requires mapping its attributes and behaviour onto a different, more tangible domain that can then be represented within the computer. This leads to the *metaphors* used in interactive applications, for example, with the *desktop metaphor* of computers such as the Macintosh, abstract properties of a computer file store are mapped onto physical referents such as iconic folders and waste baskets, allowing them to be manipulated as if they were "physical".

There are two basic approaches to visualising an abstract object: either encode all the object's facets in a single representation; or have several different representations, or *views*, each showing a single facet or perhaps a simple combination of a few related facets. Each view is an abstraction, or filter, that shows only what is important while suppressing irrelevant detail. These two approaches are shown diagrammatically in figure 3.2 where the light bulb represents the abstract object being viewed. This object has three different attributes, shown as the three primary colours of light. In figure 3.2(a) the object is being viewed through three filters, each of which only lets through one particular colour, allowing the viewer to look at each attribute in isolation. The sum of the three views shows the total set of attributes of the object. In contrast, figure (b) only lets the

object be viewed as the combination of the three colours and although it gives a good overall representation, it cannot be used to explore each separate attribute.

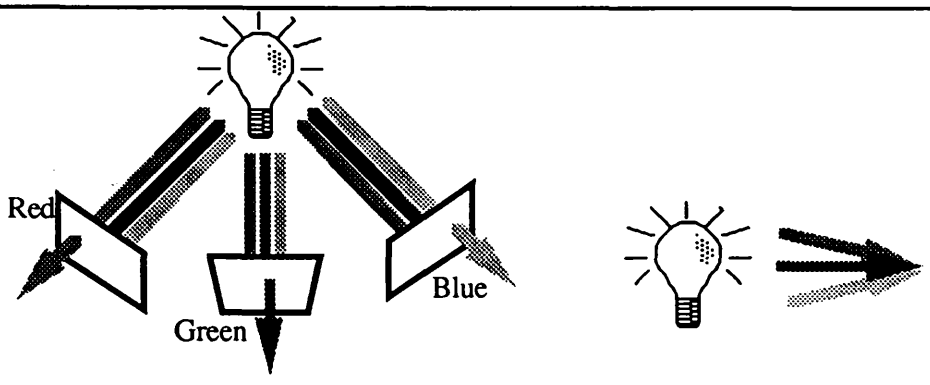


Figure 3.2: Multiple and Single Representations

The problem with a single representation is that for objects with lots of attributes the representation quickly becomes cluttered and obscure, reducing its effectiveness. A person looking at such a representation needs to locate the important information from a morass of detail. In a critical situation, such as in a nuclear reactor emergency, superimposing large amounts of information on the one representation can quickly overwhelm an operator. By partitioning the description of a complex object among several views, the amount of detail in each individual view is kept relatively small and can be dealt with in isolation. A system adopting this approach can display any number of the views at the same time depending upon the amount of screen area available and what the user wants to see. Having a choice of different views allows the user to select those of interest at a given time.

As an example of the power of multiple representations, consider the London underground railway network (the *Tube*) which is made up from a large number of stations that are connected to form a series of railway lines. These lines intersect to form a complex graph structure. The Tube has different classes of users and each has perhaps several different views of it. Each view has its own map that acts as the view, displaying a portion of the overall structure appropriate to the needs of the user. One map has a topological representation of the entire network showing the stations and lines with little geographical linkage: this would be useful to a seasoned London traveller. Another shows the geography of London with the stations and main landmarks indicated: a tourist's map. On each train is a topological map showing the ordering of stations on the line served by that train, and connectivity information to answer questions such as: *"How many stops are there before I need to change?"*. Other specialised maps include engineering plans, the layout of fare zones, and so on. Each map shows only the essential information for a

given user or application by removing irrelevant detail, e.g. an overall topological plan does not need to show every bend in a tunnel.

Software objects are also multi-faceted and it would seem reasonable to expect that a programmer would use multiple views of these objects to understand their form and behaviour more fully. These facets include (among many others) dependency relationships between components, the flow of data and control, and the organisation of data structures. However, it is only recently that display technology has reached the point where these different facets can be separated.

The mental abstractions formed by a programmer when designing a program cannot be expressed in traditional languages except unnaturally as unchecked comments that quickly become stale unless a disciplined approach is taken to maintaining comments along with the code. When it comes to maintaining the program sometime later, the programmer has to recreate these abstractions from the program text alone, and this becomes increasingly difficult as the size of the program increases beyond a trivial size. Smith (1977, p.19) suggests that this difficulty is reduced with the use of non-textual assistance:

> *"Most programmers have difficulty understanding someone else's program just given a listing of the source code. They have less trouble if they can talk to the programmer directly and get the program explained to them. They usually have even less trouble if a blackboard or other multi-dimensional medium can be used as part of the explanation."*

Workstations equipped with high-resolution graphical displays often allow multiple virtual screens, or *windows*, to appear on the screen at the same time. Sadly, this facility has, in general, only been used to support multiple user tasks (e.g. reading electronic mail while compiling a program) rather than giving multiple views of one object. However, there have been a number of notable exceptions, for example, computer-aided design tools for very-large scale integrated circuits often show two or more representations of the chip being designed. One view shows the layout of the functional blocks, another showing a colour-coded display of the layout of each physical layer, while a third shows the equivalent textual description used to control chip fabrication machinery.

Designing a Graphical Representation: an Example

The topological (or "diagrammatic") map of the London Underground mentioned above is a good example of a successful graphical representation. Its success can be gauged by the way its basic design principles have been adopted by many other underground railways around the world, and its flexibility in representing an evolving network over half a century. However, this success is no accident, and a closer look at the evolution of this map will illustrate just how difficult it is to design a useful, clear representation of a complex entity.

The London Underground map was first proposed by Harry Beck in 1931 with his first public version appearing in 1933. Since that first version, modified versions have appeared roughly at the rate of one per year. Although many of these modifications were needed to handle changes in the Underground itself (e.g. new stations being opened), a large number were the result of changes to the basic design, for example, different symbols for line intersections. It was not until circa 1950 that Beck was finally happy with the map's design.

Despite the numerous design iterations involved in producing the map it still contained flaws. For example, the stations of Wimbledon and South Wimbledon are less than one hundred metres apart but Beck's map separates them by what seems a long distance. A later revision of the map by a different designer tried to rectify the discrepancy by moving the stations closer together on the map. Unfortunately, the ramifications of this change resulted in a drastic alteration in the map's appearance and was not considered a success: the subsequent version of the map reverted to Beck's original layout, warts and all.

This example illustrates the importance of iterative design and evaluation in the creation of a successful graphical representation: it is foolish to expect the representation to be perfect from the outset. Furthermore, compromises may have to be made between total accuracy of the representation and the tractability of its generation: users may prefer a simpler form to a complicated one even if it is not strictly correct. These general lessons must be remembered when designing the graphical representations of the highly abstract entities encountered in software development.

3.2.5 The Potential of Colour

Colour display hardware has, until recently, been an expensive and hence uncommon peripheral for a computer. Although not yet widespread, colour workstations are now becoming cheaper and more accessible, with the promise of colourful user interfaces. Unfortunately, in many cases, colour is a solution in search of a problem: a great deal of the work into using colour has been motivated by having the technology available rather than by any sound need.

The use of colour in user interfaces presents several problems. Firstly, the developers of such interfaces are often not aware of basic graphic design rules with the result that interfaces make poor use of colour. Although design guide-lines do exist (e.g. in Shneiderman 1986) they are often not heeded. There is also the physical problem that about eight percent of males have defective colour vision, with the serious risk that any information represented using colour alone could be lost to such a user. Perhaps the most subtle problem to be overcome is with colour associations, e.g. red is commonly associated with danger and using it to indicate something else may still invoke anxiety in a

user. This problem can be overcome by ensuring that colours are used sensibly for a particular domain.

However, despite these problems, colour does allow an extra dimension of information to be encoded into a visual representation. All objects related in some way could be displayed using the same colour, e.g. values with the same type. Colour can also be used to highlight particular features in a complex display, e.g. error states shown in red. Systems may also use sequences of colours to highlight ordering of information, e.g. a spectrum of colour can indicate an ordered array of numbers.

3.3 Visualisation and Programming

Many different systems that exploit graphical technology have been developed for a variety of domains, e.g. control displays in nuclear power stations, computer-aided design packages, etc. In this section the discussion will be restricted to representative systems involved in the programming process, and some influential systems from closely related fields. The programming process is taken to span the specification, coding, debugging and maintenance phases of the software life-cycle.

Two terms have entered common usage in discussing the involvement of graphics in the programming process: *Program Visualisation* and *Visual Programming*. Despite their similar names, these terms refer to different aspects of programming. In his survey paper, Myers (1989) regards Program Visualisation as illustrating some aspect of an existing, *textual* program or its run-time execution. This seems overly restrictive as it excludes those programs that have a non-textual representation. Visual Programming on the other hand, *"refers to any system that allows the user to specify a program in a two (or more) dimensional fashion"*. Raeder (1985) gives a survey of current techniques involved in visual programming.

Myers (1989) presents a taxonomy of contemporary, visually-based systems by first partitioning them into either visualisation or programming categories. Program visualisation systems are further classified according to whether they illustrate the code or the data of a textual program or its execution, and whether the display is static, shows a series of snapshots or is dynamically updated to reflect the execution of the program. Visual programming systems are classified according to whether programs are compiled or interpreted, and whether they support programming-with-examples.

This chapter presents an alternative taxonomy by categorising systems according to their intended rôle in the programming process, complementing the one given in Myers (1989). The taxonomy starts at the end of the programming process when a program already exists and then moves through coding towards the specification phase, discussing representative

systems intended for each phase. This retrograde order is used as it illustrates how visual programming systems build on the techniques developed for program visualisation.

3.3.1 Understanding a Program

Improved Static Representations

A programmer's desk is often submerged in reams of paper listing the source code of programs, past and present. This static representation has many benefits: it is a permanent record; large parts of the program can be viewed simultaneously; textual and graphical annotations can be easily added using a red pen, for example. Although large-screen workstations can go some way to supplying these features they are not so flexible and cost a great deal more. In addition, textual programming languages have evolved (at least implicitly) to match these features: the important global declarations appear at the start of the listing, indentation of lines can show the nesting of program blocks, and the common define-before-use principle helps to locate components within the listing. However, it could be argued that such ordering is merely for the benefit of the compiler and may actually hinder the top-down *development* of modular code by requiring the top-level modules to appear at the end of the listing.

Unfortunately, until recently, printer technology was relatively crude limiting output to a limited alphabet of characters found on a typical typewriter. With the availability of laser printers capable of advanced typographic features such as arbitrary graphics and multiple fonts and character sizes, it seems sensible to exploit these features to improve the static representation used in hard-copy listings. Baecker and Marcus (1986) describes just such a *pretty-printer* for the C programming language, presenting before and after examples of source code listings. Their (rather ad hoc) techniques for improving the representation include graphical highlighting of comments; indicating any abrupt changes in control flow (e.g. RETURN statements); graphically delimiting the body of a routine; and including contextual information such as file names, page and version numbers, and dates. They argue (but without presenting firm evidence) that the increase in the length of listings is more than compensated by the improvement in program understanding and hence maintainability. They conclude by suggesting an extension of their work to interactive program visualisation.

Algorithm Animation

In order to understand a non-trivial algorithm it is not usually sufficient simply to read its static representation as a program listing. A proper understanding is only achieved by watching the program execute, by observing the flow of control and the changes to various data structures. *Algorithm Animation* is the dynamic visualisation of the run-time behaviour of an executing program. The intention is that the visualisation will give insight

into how the underlying algorithm works by displaying a dynamic representation of its execution rather than the programmer needing to imagine it given a static, textual program listing. As Sutherland (1966) states: *"Being able to see a program run gives one a grasp of detail that is hard to obtain in any other way"*.

Giannotti (1987, p.311) presents experimental evidence that algorithm animation *"is of greater help to novices in understanding algorithms than traditional textual methods"*. Typically, users of such systems will be program maintainers, novices learning about algorithms, developers of new algorithms, all of whom are trying to understand the workings of an existing program. The programs that have been successfully animated are usually quite small due to difficulties in visualising large programs (as discussed later in this chapter).

In a traditional programming environment, the dynamic behaviour of a program is usually illustrated by instrumenting the program with extra statements to print messages and the value of important variables. Such results are difficult to interpret because of the large quantity of data produced, lack of integration with the actual program text, and the low-level representation used to print data values.

Recently, much research has been done on improving this situation by using graphics workstations. Perhaps the best attempt so far is the BALSA system which is described by Brown and Sedgewick (1984). By developing a special-purpose system tailored to their needs, real-time animation of a program's execution is possible using multiple, textual and graphical, ad hoc representations. Colour is often used in the graphical views to illustrate the temporal behaviour of an algorithm. Implemented views include textual highlighting of the currently executing source code statement, procedure invocation histories, graphical display of data structures such as trees and arrays, as well as novel geometric representations of program attributes.

BALSA has facilities for creating scripts to control the animation, and giving interactive control over program execution. The creation of an animation has several stages with a different class of user involved at each stage. These users include the algorithm designer, a specialist animator, and a scriptwriter who prepares a "dynamic book" describing the algorithm to the eventual reader. To create a "good" animation requires specialist knowledge of the algorithm to identify interesting events in the executing program. Whenever an interesting event occurs, the animation system causes the displays to be updated to reflect the new state of the program. Future developments of BALSA are intended to reduce the time taken to produce a new animation by increasing the library of available views and tools to simplify the identification of interesting events. It is also intended to use micro-computers such as the Macintosh as the host environment for animations thereby making it more widely available.

The PV prototype system described in Brown et al. (1985) supports the manipulation of static and dynamic diagrams of programs, with emphasis on understanding a program through the dynamic visualisation of its data structures at levels of abstraction specified by the user. Given the diversity of data formats, PV was designed to simplify the creation of new dynamic visualisations by the user, using animation templates from a library. The system also simplifies the binding of graphics to code by avoiding the need for the manual insertion of calls to library routines into the program, as with BALSA. Bindings are instead created using a special graphical editor to link graphics and code.

Debugging

Debugging is similar to algorithm animation except that the purpose is to understand erroneous behaviour in a program's execution and then locate and correct the fault. As with algorithm animation, the program already exists, usually in a textual language but can be of an arbitrary size. Most high-level programming language implementations have a tool called a *source-code debugger* to assist in locating faults in any program written in that language. Debugging differs from algorithm animation in that the program does not need to be specially instrumented for it to be viewed, the compiler can inject the necessary support for the debugger automatically. Debuggers must be general-purpose, not needing to be tuned to the program under observation. Since the debugger has no specialist knowledge about the program, the visualisation of an executing program is quite crude.

The programmer can use the debugger to stop the execution at certain points in the program, examine and perhaps change the values of variables, resume execution or step through the execution a statement at a time to observe the low-level behaviour at a critical stage. Traditional debuggers are text oriented with little support for displaying complex data structures in a useful manner, such as those involving pointer networks. Unfortunately, program errors are often in just such structures.

Several systems have been developed in an attempt to simplify debugging through improved visualisation techniques, particularly in displaying complex data structures. They were motivated by the observation that real programmers manually draw data structure diagrams when trying to understand or debug a program, and simply tried to automate the process.

The INCENSE system described in Myers (1983) supports the interactive display of data structures used by programs written in the Mesa programming language. INCENSE allowed variables of all types, including record and pointer structures, to be displayed using a variety of representations, including ones created by the user for the special display of data abstractions. Unfortunately, the display of an even moderately complex structure took over thirty seconds to generate, which reduces its appeal in an interactive environment. However, the main reason for INCENSE failing to be adopted was the poor

integration with the rest of the Mesa programming environment. In particular, it was not part of the standard debugger and had no facility to edit data structure values.

The KAESTLE editor described in Böcker et al. (1986), was developed to display and edit LISP list structures, and went some way to overcoming the limitations that prevented INCENSE from becoming a standard debugging tool. As they state:

> *"For a widespread use it is of critical importance that these [visualisation] tools are tightly integrated and easily accessible within the general programming environment"*. (p.48).

The KAESTLE editor was integrated with the standard debugger and trace package which allowed for snapshots of structures to be taken. Special layout heuristics were developed to ensure that a display was generated quickly, taking into account the amount of screen area available, and which parts of a large structure were deemed important. Once a display was produced the user could clean it up, e.g. by moving parts of the structure, removing some parts and expanding others. Once displayed, the editor allowed the structure to be modified by adding or removing atoms and links and changing stored values.

INCENSE and KAESTLE both produced a static display of a data structure with little attempt to animate the display to illustrate dynamically changing data values. One recent system that derives great benefit from a dynamic display is the Transparent Prolog Machine (TPM) described in Eisenstadt and Brayshaw (1987, 1989). The TPM was designed to assist both novice and expert Prolog programmers by providing a graphical trace and debugging facility. The user can control, among many other aspects, the level of detail displayed, and the speed and *direction* of execution. The complete execution state is displayed through dynamically updated AND/OR trees whose nodes are boxes graphically depicting the status of each goal and clause. Effective use is made of colour to highlight patterns in the execution. The authors discovered that programmers were able to exploit these gestalt patterns to get an overall feel for the behaviour of their program: something that would be almost impossible with conventional program traces.

Flexible Presentations

With the systems discussed so far for visualising a program's data structure, the display has been determined by the limited number of representations used by the system. The user could not specify a new representation except by augmenting the set of views. Some work has recently been done to allow one or more representations to be used to display an entity, to be specified along with the program. Such specifications are often called *Presentations* and have been investigated as a way of visualising abstract data types, for example in Powell (1987) and Szekely (1987). However, the results are mostly theoretical although Powell has presented an implementation of the basic ideas.

3.3.2 Coding

The coding phases of software development involves taking a detailed, non-executable description of what a computer system should do and convert it into an equivalent, executable description or *program*. Many researchers have investigated the application of visualisation techniques to the coding of programs: a field that has become known as *visual programming*.

Visualisation systems to help someone understand a program usually involve little input from the user other than commands to control the display, or to select objects to be displayed. It is a general belief that a representation that is successful in visualising a program and its execution, will also be a good representation for writing those aspects of the program shown in the representation, given suitable interaction techniques. Many of the systems to be described in this section extend good manual or automated graphical representations allowing them to be used in programming.

The taxonomy presented in Myers (1989) imposes the artificial restriction that only systems visualising programs written in a conventional, textual manner will be regarded as Program Visualisation, thereby excluding all programs with graphical form. The taxonomy presented here takes the more general view that any program can be visualised regardless of its final representation. Visual programming systems are thus seen as extending program visualisation to include a programming facility, while retaining the ability to let the user explore an existing program.

Visual programming systems can be partitioned according to whether they assist in the development of programs whose final form is a text-based language, or instead break away from tradition to explore new possibilities offered by exploiting graphics in a more novel way. These two camps will be discussed separately using representative systems to highlight the important issues.

Traditional Approaches

An on-going process in programming is the development of new and better tools to make programming itself simpler, less laborious, with improved detection and correction of errors. The recent developments in graphical technology have enabled the creation of a whole new range of tools to assist the construction of programs written in traditional programming languages. However, the problems of trying to support large-scale programming with graphical techniques have still to be solved and to date no system has been produced that can cope with the size of programs being tackled by industry.

Much greater progress has been made in using graphical techniques to help teach novices to write programs in traditional languages, usually Pascal given its common rôle as a teaching language. A variety of approaches have been tried. Pong and Ng (1983)

describe the use of Nassi-Shneiderman diagrams in the PIGS system. These diagrams are simply boxes placed around sections of linear Pascal code to reflect the control structure of the program. With this dependence on textual code, PIGS' claim to be a visual programming system is diminished. A similar situation arises with the OMEGA system, described in Powell and Linton (1983), which also uses a limited graphical vocabulary as a façade to a basically textual programming system.

The PECAN programming environment generator, described in Reiss (1984), can produce a system capable of showing multiple, consistent views of a Pascal program using textual and graphical representations, with programs built by textual or graphical editing. These views show the state of the program and its execution in traditional ways, for example, as textual source code, as a flowchart, the stack of current procedure invocations, etc. As the program executes, these views are updated, for example by highlighting the currently executing statement. PECAN merely automates the sort of representations used manually during programming, without offering any new view that exploits the expressive power of graphics to improve program understanding.

Edel (1988) discusses a graphical interface to LISP called TINKERTOY which uses icons and flexible links to represent programs and data structures with the intention of removing the clutter introduced by LISP's textual syntax. Programs are constructed by joining together iconic structures using simple but powerful graphical editing commands, with functions defined using a *circuit diagram* approach. Unfortunately, these diagrams become unintelligible for all but the most trivial of structures. This is due to the structures used in LISP, and also the rather poor graphical form used to represent operations and links. Other problems with TINKERTOY include the slow generation of graphical layouts; anomalies in executing programs because of the need to first translate graphical programs into LISP code; and poor support for input-output operations.

Novel Approaches

Considerable research has been done to develop new graphical representations and metaphors for programming. Generally, the systems produced have been experimental, capable of handling only relatively small programs, with the intention of simply trying out a new approach. This section discusses some of these developments using representative systems as examples.

Perhaps the first visual programming system was the Graphical Program Editor (GPE), described in Sutherland (1966), which influenced the design of many future systems. GPE uses techniques developed in computer-aided design systems for building electronic circuits, with components representing functions joined together using "wires" to form a data-flow network. The termini of a component were given types either explicitly from a menu or inferred from its connections, thus ensuring that only type-correct programs could

be constructed. The graphical program is compiled into machine code and executed, with the facility to examine values dynamically by attaching *test probes* to the circuit diagram of the program. Sutherland concludes by noting that GPE suffers from the perennial problems of how to handle large programs graphically, and in executing them efficiently, but has the foresight to recognise that systems such as the GPE would be very useful in understanding the behaviour of parallel programs.

Christensen (1968) describes the AMBIT/G visual language for manipulating and creating directed graphs, supporting research into graph theory and syntax analysis as well as the development of algorithms for non-trivial operations such as garbage collection. The programmer creates nodes in the graph with shape being used to indicate typing information. Templates are drawn to show how nodes may be legally connected, with graphical pattern matching used to define transformations on a graph structure. A graph entered as input to the system would be transformed using these graphical rewrite rules. Unfortunately, the lack of suitable graphical hardware prevented the implementation of a completely graphical programming system.

A common tool (although now looked upon less favourably) in assisting the writing of programs was the flowchart. It was therefore inevitable that visual programming systems were developed that use the flowchart as their underlying representation. Some, like the PIGS system described above, merely decorated textual code with flowchart symbols to highlight the control structure in the program. The PICT system described in Glinert and Tanimoto (1984), adopted a more novel approach placing (rather confusing) icons inside the flowchart boxes rather than text, with colour being used to represent variables and data paths. Unfortunately, the language supported by PICT is so limited that even novices found it restrictive after a short time.

The first system to combine a graphical representation with the programming-with-example paradigm was PYGMALION, described in Smith (1977), and the inspiration of many subsequent systems. PYGMALION was designed to act as "dynamic scratch paper" allowing the user to experiment with ideas by drawing and manipulating iconic structures through pointing and issuing commands from a menu. The system has a *Remember* mode where it "watches" the user edit a graphically-displayed data structure and records the operations as a procedure capable of operating on the same kind of data. An interesting difference with other systems is that a PYGMALION program does not have a static representation. Since programs are created by a sequence of graphical editing actions, the proper representation of a program is a sequence of displays: a movie. Since the control flow is now spread along the time dimension, individual snapshots of the state of program execution are now simplified, especially when compared with systems such as AMBIT/G where the control information was wrapped up with transformational details in one static representation.

A PYGMALION procedure can be invoked on graphically-represented input data using the *Display* mode. Rather than force the use of a fixed graphical vocabulary, if the user creates the graphical images then the amount of translation required to map between the mental and computer representations of a solution is reduced. For example, a problem involving an electronic circuit would use standard images of electronic symbols. Like most visual programming systems, PYGMALION is slow in executing programs written graphically, and is limited to handling only relative small programs as the display quickly becomes cluttered.

The THINKPAD system described in Rubin et al. (1985), extends PYGMALION to define ADTs using the programming-with-example paradigm, with traditional programming replaced by graphical editing. ADT operations are demonstrated by editing example data represented graphically, with the facility to place constraints on the data structure that must be observed by the operations. These constraints are used to enforce strong typing, and express dependencies and relationships among the fields of the data structure. The constraints are specified using a simple textual editor and are a simplified version of the constraint mechanisms of THINGLAB described in Borning (1981, 1986) and used in general-purpose programming. The authors of THINKPAD argue that this style of graphical programming-with-example is closely related to the declarative style of Prolog. The graphical programs created with THINKPAD can be translated in Prolog statements and then executed. There is currently no way, however, to display the results of this execution graphically, although future plans include the development of such a facility. THINKPAD is also write-only in that it is incapable of visualising any of the programs it has created.

The PROC-BLOX system described in Glinert (1987) uses graphics to give a geometric representation to syntactic structures in Pascal. Program constructs are shown as tiles resembling jigsaw pieces. Programs are constructed by joining these pieces together to form a syntactically correct program, with pieces fitting together only if they have complementary shapes. By requiring only the recognition of geometrical shapes, the jigsaw metaphor abstracts away the details of syntax known to be a major hurdle for novice programmers.

3.3.3 Software Specification

Specification is perhaps the most interesting and critical part of the programming process. It is interesting because it uses a programmer's creative talents to an extent greater than in any other phase. It is critical to the entire development process because incorrect decisions or errors introduced in this phase will require very expensive and time-consuming repair when discovered during the later phases of system development or use.

"The hardest part of the software task is arriving at a complete and consistent specification, and much of the essence of building a program is in fact the debugging of the specification."

Brooks (1987, p.16)

Given the importance of specification, it is therefore surprising that relatively little work has been done in providing mechanical support for this phase when compared to the vast amount of support for the coding phase, for example. However, it should always be borne in mind that computerised tools can only *assist* the (human) designer, being incapable of giving creative advice. Typically, computers will handle the tedious administrative tasks such as maintaining a record of the specification's evolution or perhaps performing certain rote operations.

Software specification takes a set of customer requirements† as its starting point and constructs a specification (or more precisely, a family of specifications, each at increasing levels of detail) of what the software should do, and then how a solution to a particular problem may be implemented on a computer. Within this abstract notion are many different styles of approaching the transformation, each with different implications on how a computer may support the process. The different styles may be split into *informal*, *semi-formal* and *formal* specification.

With informal specification the specifier uses techniques acquired through experience in building previous software systems (this is the craft approach mentioned in chapter one). This often involves the use of personal graphical notations such as note pad or blackboard doodles which offer little opportunity for being incorporated within a computerised tool because of the unrestricted, free-form nature of the notation.

Semi-formal specification adds structure to the informal approach by providing guide-lines and regular notations that improve the likelihood of creating a successful specification. It is interesting to note that many of the approaches in this group are based around graphical notations, for example Petri-nets and data-flow diagrams. Clearly, graphical editors can be used to simplify the creation and modification of such notations, especially if the editor has knowledge about the particular notation involved. As well as supporting the drawing of design diagrams with a formal, semantic meaning, the editor could ensure that any guide-line violations are reported and then corrected, and perhaps even automatically convert the diagram into a skeletal program. An example of such a system is PEGASYS, described in Moriconi and Hare (1985). With PEGASYS, the pictures used to represent a

---

† This discussion conveniently ignores the considerable problem of arriving at the user requirements in the first place. Typically, problems arise due to the difficulty in understanding what the customer wants and the customer not being able to articulate their needs.

design exist within a logical framework. The user manipulates these pictures using graphical operations with the system ensuring that the constraints of the underlying logical framework are not violated. The authors found that the graphical interface made the development of a formal design more palatable to program developers.

As discussed in chapter two, formal specification is usually textual or symbolic in nature, making it better suited to support from conventional text editors rather than graphical tools. Many different techniques have been proposed for the formal specification of software systems, but lack of mechanical tools has limited their appeal to developers of large systems. However, much work is currently underway to provide tools to support formal specification on an industrial scale. Unfortunately, these tools are usually theorem provers or for supporting the execution of specifications, with little attention paid (so far) to using graphical techniques.

One proposed system for using graphical techniques to specify software systems algebraically will use the Programming-by-Generic-Example idea introduced in Goguen (1986). The proposal is for a graphical interface to the OBJ2 equational programming language which has a computational model based to rewrite rules (see chapter two for details). The user will be able to create new rewrite rules by direct manipulation of *generic* examples of data. The generic example permits the direct manipulation of an object representing an arbitrary part of a data structure, for example the k^{th} element of an array. This overcomes the problem with other example-based programming systems in trying to infer a general case from a specific example. Unfortunately, this graphical representation of such generic examples is limited to linear data structures, and has yet to be incorporated within the OBJ system.

3.3.4 Other Related Work

Graphical techniques have been adopted and developed by a number of applications for many domains in an attempt to improve on textual methods. Although not basically concerned with the programming process per se, they demonstrate new ideas that could be successfully adopted by the next generation of programming tools. This section discusses the ideas behind several of the more inspirational systems.

Graphical examples are commonly used in teaching to illustrate an abstract concept by giving a concrete example of what is going on. Many systems have been developed that use interactive computer graphics to simplify the generation and manipulation of these examples within a particular domain, the most striking example being the Alternate Reality Kit (ARK) described in Smith (1987). ARK is an environment for creating animated simulations of physical situations. The user is presented with a computerised world with objects that can be manipulated using a mouse-operated "hand". With the hand, the user can pick up objects, throw them, create new ones, and send *messages* to them by pressing

simulated *buttons*. All displayed objects (including non-physical ones, such as computational objects imported from the underlying Smalltalk universe) are subject to physical forces, such as gravity, that the user can alter to create *alternate realities*. The adoption of this physical metaphor means that the learning time for novices is kept short without compromising on power.

A similar approach was used in the Programming-by-Rehearsal system described in Finzer and Gould (1984) which uses a theatre metaphor to simplify the construction of educational software. Programming consists of moving *performers* around on a *stage* and teaching them how to interact by sending *cues* to one another. The theatre metaphor provides real-world referents for the computational model used by the Smalltalk system where computation consists of objects sending messages to each other. Experiments with Rehearsal World have shown that teachers with no experience of programming can create complex and interesting material in a very short space of time, working entirely within the confines of the theatrical metaphor. This confirms the ARK findings that choice of a suitable metaphor can make programming accessible to a much wider audience by appealing to their knowledge about how objects behave in a familiar world, such as the theatre.

Another area of research to use graphics to improve on textual methods is user interface design. Workstations with graphical displays offer a host of new possibilities for making the power of the computer more accessible to a wider user population. The intention is to liberate the computer from the idiosyncrasies of traditional command language interfaces by making human-computer interaction more physical in what is now termed *Direct Manipulation* (Hutchins et al. 1986). The pioneering work in this area was done at Xerox, extending research into the Smalltalk environment. This culminating in the design of the STAR user interface that introduced the now ubiquitous desktop metaphor (Smith et al. 1983). In command language interfaces such as the UNIX shells, it is possible to extend and customise the user interface by writing scripts by combining existing commands. Such a facility is extremely powerful and Halbert (1984) describes an attempt to add one to the STAR user interface. The user creates a program by giving an example of what it should do using the standard graphical commands available at the user interface, with the system recording the sequence of actions as a static textual representation. However, the visual programming-with-examples mechanism is not ideal for describing the control structure which had to be added by editing the textual representation of the program.

3.3.5 Problems with Visualisation

Too Much in Too Little Space

Most of the systems described above reported a number of problems peculiar to the graphical medium. The main one was the way visual representations rapidly become

cluttered and difficult to understand as the program being visualised increases in size: the so-called *spaghetti problem.* This has restricted the application of such systems to domains where smaller programs are used, such as teaching novices how to program or in understanding the behaviour of an algorithm. One cause of this problem is the explicit representation of relationships between components as graphical connections between parts of a picture. Edel (1988, p.1113) regards these connections as *"wasted space"*, claiming that they do not exist in textual languages. Such connections do exist in programs but they are usually hidden in conventional code. The absence in textual languages of explicitly stated relationships between components means that programmers are often unaware of the ramifications of a change leading to the common situation of introducing new bugs in trying to fix an old one. Graphics are particularly good at displaying relationships between objects, and a program representation that makes visible the dependencies that a programmer would otherwise have to discover for himself is a distinct advantage.

Various graphical techniques are available to overcome the spaghetti problem. The obvious approach is to reduce the amount of information displayed. This can be done by the system using heuristics to decide what is important in a given context and removing the unimportant: as in the KAESTLE editor. A better user interface would allow the user to influence this filtering process. The *Fisheye* mechanism described in Furnas (1986) uses a combination of user-assigned and system-assigned importance ratings to control the display. Rather than simply remove unimportant information, the level of detail shown for an object depends on its current importance, for example, currently unimportant parts can be shown smaller or banished to the periphery of the display. This provides a balance between local detail and global context.

Other more common ways of handling large graphical displays is to use techniques such as *zooming* and *panning* of the display. Zooming allows the user to close-in on interesting detail or pull back to show the overall picture. If multiple windows are available, one window could be used to show a magnified view of a certain part of the structure with another showing the global perspective. Panning allows the user to move a window over a much larger virtual display that is too big to fit on the screen. It can be usefully combined with zooming to produce a very flexible way of viewing a large display.

An alternative way to control the amount of information displayed is to allow an entity to be observed at various levels of abstraction with the user choosing the one most suitable for a given situation: as in the PV system described by Brown et al. (1985). For example, a program can be viewed as a set of modules at a high level of abstraction and as machine code instructions at the lower end. When trying to understand module dependencies the machine code display will simply confuse rather than enlighten. Hansen (1971) calls the substitution of a simple symbol for a complex object *holophrasting*. For example, if a

program's modules are represented by a set of named boxes then each box is acting as a holophrast for the corresponding program fragment. Each holophrast may be expanded into a representation of the next level of detail of the underlying structure; this in turn, may be composed of other holophrasts.

Poor Interactive Performance

Many of the visualisation systems described above were restricted to the display of relatively simple data structures such as Arrays, especially if the display is dynamic. With systems capable of displaying arbitrary structures, such as INCENSE, the time taken to generate a display is often prohibitively long for interactive use. These systems try to produce a pleasing display automatically, a task shown in Johnson (1982) to be NP-complete in the general case for structures such as trees. Instead of trying to do everything automatically, a more sensible approach is to produce a "quick-and-dirty" display and let the user tidy it up if desired. With this approach, the user is not frustrated at having to wait for the display to be generated and can customise the display using his own subjective aesthetic criteria. This approach is used in the KAESTLE editor.

The problem of displaying large data structures is compounded if changes in the data are to be shown as a dynamically updated display. Careful design of the animation system is required to ensure that displays are updated in a natural and timely fashion.

A common complaint from expert users of interactive systems is that graphical input techniques are too slow, for example, entering numbers by using the mouse to click on a graphical representation of a numerical keypad is much slower that entering numbers from a real keypad. This is compounded by the time required for the user to moving his hand to the mouse, perform the action and then return to the keyboard. However, it is not a simple case of textual input always being better than graphical input. Consider the case of selecting a part of a data structure for more detailed display. Textual input would require the user to give the name of that part, which in a complex structure involving pointers, may be quite long. There is a high risk that the user will mis-type the name or may even give the wrong name. A graphical approach would allow the user to select the part simply by pointing at it. Clearly, both approaches are useful and a well-designed system would allow the use of the one most appropriate in a given situation, e.g. even the graphically-oriented Macintosh computer provides *accelerators* that allow common commands to be issued from the keyboard rather than from the pull-down menus.

Incomprehensible Graphical Representations

Programmers are used to dealing with static, textual representations of programs: a skill requiring many years of training to master. It is therefore not surprising that novel visual representations are found to be difficult to understand on casual inspection. Myers (1989) claims that graphical programs are often hard to understand once created and difficult to

debug and edit, particularly if they get to be a non-trivial size. However, this argument can also be brought against textual programming languages, and illustrated by the problem in maintaining someone else's code. The case against graphical representations is not a fair one since programmers are generally not well acquainted with them. This is nicely summarised by Shneiderman's confession:

> *"I found it difficult to think about information problems in a visual form, but with practice is became more natural. With many applications, the jump to a visual language was initially a struggle, but later I could hardly imagine why anyone would want to use a complex syntactic notation to describe an essentially visual process".*
>
> Shneiderman (1983, p.66)

3.4 Summary

Advances in display technology offers the opportunity to have user interfaces capable of displaying high-quality graphical images as well as text. Many visually-oriented applications have been developed for numerous domains that exploit this new facility.

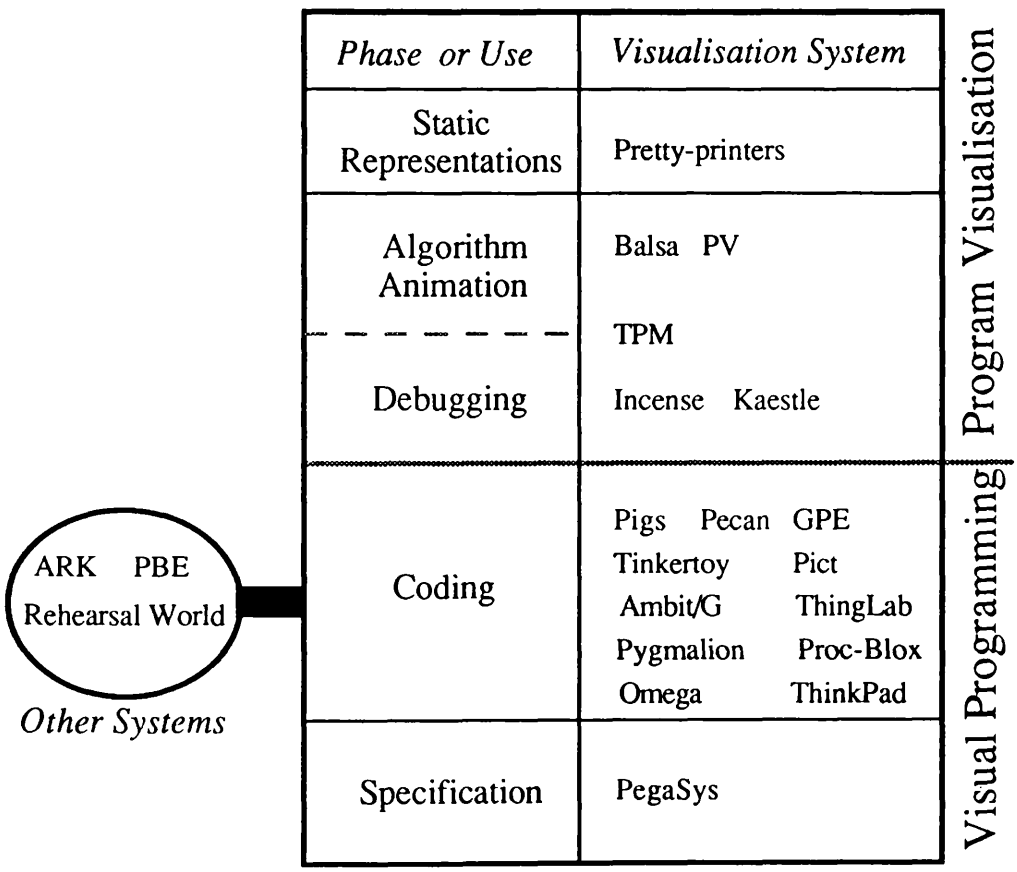


Figure 3.3: Classification of Visualisation Systems in Programming

This chapter presented a new taxonomy of a representative sample of such systems that were designed for use in programming, by considering their rôle in the software development process. This is summarised in figure 3.3.

Perhaps the most striking feature of this figure is the concentration of systems in the coding category with fewer systems supporting earlier or later phases of software development. Although the survey of visualisation systems in this thesis covers only the most important developments, it is nevertheless a reasonable sample of the distribution of systems among the different phases.

Program Visualisation uses graphics to give a model of the internal state of an existing program and its data structures. By making this normally hidden state visible by means of suitable (often dynamic and colourful) graphical representations, the intention is to allow the behaviour of the program to be understood. This has obvious applications in teaching novices how algorithms work, and in debugging faulty programs.

An extension of this visualisation allows the program to be built using graphical techniques. The program can either be represented using a conventional textual form or in some graphical form. Similarly, the specification of the program can either automate some existing manual technique such as flowcharts or experiment with more novel approaches. Commonly, visual programming systems let the user create a program by manipulating examples of the expected data with the system recording the user's actions. There is evidence that mental representations are often visual and so giving abstract objects a graphical form makes them easier to manipulate and comprehend.

The taxonomy concludes by looking at the little work done or proposed for supporting the design and specification of software. This area is interesting because it has been almost ignored by developers of visual tools despite its desperate need for computer tools. The VISAGE system described in the following chapters belongs in this area.

Since visually-oriented systems have been developed for many domains, some of the more important systems from these domains were analysed for their possible contribution to future programming systems.

The application of graphical techniques to the programming process should not be regarded as an attempt at replacing textual programming. A more pragmatic stance is taken with graphical representations seen as complementing textual ones. Workstations can display multiple views of a program simultaneously, with each showing a particular aspect in a way likely to give greatest insight to the programmer. The programmer can also edit the program using the most appropriate view at a given time. Visualisation is at an early and exciting stage in its application to software development, and the chapter concludes by describing some of the problems needing to be tackled, with suggestions for possible solutions.

Chapter Four
The Visualisation of Abstract Data Types

4.1 A Marriage of Convenience

The principal contribution of this thesis is the bringing together of two previously separate domains: the formal specification of ADTs (as discussed in chapter two) and the use of computer graphics to improve the development and understanding of software (as discussed in chapter three). Why should these two domains be brought together and what benefit would come from their union?

Chapter two discussed how formal specification techniques could be used to ensure the correctness of software, and in particular the design of ADTs. Unfortunately, the formalisms involved demand a mathematical sophistication greater than is commonly possessed by software engineers, and the esoteric notations do not encourage novices to start using formal methods as an everyday development tool. If a way could be found to overcome these barriers then this would act as a catalyst to the increased use of formal methods in software development. The results and experiences of using visualisation systems such as those discussed in chapter three suggest that tools exploiting the power of computer graphics would be one way to make formal specification more palatable to the ordinary software engineer.

Incredibly, although many visualisation systems have been developed to support the coding and debugging phases of software development, there has been no attempt to visualise the *formal* specification phase of the process. This is a puzzling deficiency given the importance of the specification to the success of the entire system, and the highly abstract nature of the subject. It is the goal of the research described in this thesis to investigate how visualisation techniques can be applied to the formal specification of ADTs and then to evaluate whether their marriage is a success.

To test the thesis that the visualisation of formally-specified ADTs can make them more acceptable and understandable to novice users involves building a demonstration system that can be evaluated using subjects representative of the intended user population.

4.2 Overview

The design and development of VISAGE was done in four stages. This chapter discusses the design principles under-pinning any interactive, graphical system intended to support

the algebraic specification of ADTs. It also considers the foundational issues involving in building such a system.

The second stage, discussed in chapter five, developed a visualisation system to help a user understand an ADT given its formal, textual specification. This phase investigated the need for different views of an ADT, experimented with different layout strategies and developed the basic architecture of the overall system. This basic system was not particularly interactive, only allowing the user to select a particular ADT to visualise and then perform cosmetic changes to the display.

The third stage, discussed in chapter six, extended the first system to allow specifications to be created and manipulated through textual and, more interestingly, graphical editing. This visual programming capability included the design of an integrated tool for experimenting with example data values to check that a specification behaves correctly.

Chapter six also describes the fourth stage that builds upon the results of the previous ones by developing a graphical tool, called the *PLAYPEN*, based around a term-rewriting engine. The PLAYPEN allows users to create graphically example data and have them transformed using the equations of a selected ADT.

4.3 Is a Graphical Interface Appropriate?

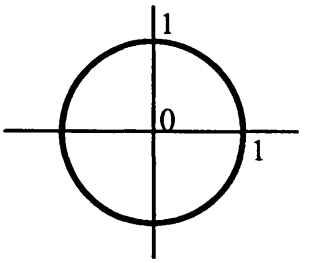
As one would expect, the formal specification of software is being advocated and developed primarily by mathematicians who have acquired the skills and intuitions for manipulating the highly symbolic representations involved. In contrast, practising software engineers typically do not have such skills or intuitions and to make formal specification a working tool for them requires a considerable investment in training or improved support tools or better, a combination of the two. Given the need for skills in manipulating mathematical descriptions of an ADT, why involve visualisation?

In traditional mathematics, pictures are usually regarded as a medium for illustrating a point defined rigorously in a symbolic notation. Although this demonstrates the power of pictures to communicate ideas in a clear and readily understandable way, pictures are not usually regarded as having sufficient precision to be used to communicate the concept on their own since the flexibility of a pictorial vocabulary results in pictures that are often confusing and easily misinterpreted. However, if a formal mapping exists between the displayed pictures and the underlying formal model, then pictures created by the user can have the same precision as a symbolic description.

Unfortunately, as is common with the emergence of alternative approaches to a task, there is considerable resistance by mathematicians to the use of visualisation techniques in mathematics. This resistance is natural given the drive in mathematics in recent centuries to reduce the dependence on visual notations, e.g. in geometry, in favour of more

symbolic notations as a way of facilitating the manipulation of formal definitions. However this coup d'etat by symbolism has pushed mathematics further away from the inherently visual representations used in the human mind for solving problems.

For example, the symbolic expression $x^2 + y^2 = 1$ is considerably divorced from the mental, visual representation of the unit circle, and although it makes certain manipulations easier, it does not reveal the symmetry of form as readily as the graphical representation shown on the right. It is ironic that the word *theorem*, with its association with symbolic statements, is derived from the Greek verb *to look*.



The majority of arguments against the adoption of visual images in mathematics are based on the premise that the intention is for visualisation techniques to usurp the throne currently held by symbolic techniques. This premise is in error: as has been argued earlier in this thesis, visualisation should instead be seen as a way of complementing symbolic notations.

The visualisation tools discussed in chapter three can be arranged on a spectrum from the general-purpose to the specific. The extremes are neatly characterised by two classes of tool: debuggers and algorithm animation systems. Debugging support tools such as INCENSE (Myers 1983) must work with any program that is legal within its particular environment. To allow such generality, its graphical representations cannot exploit special features of a particular program under observation to enhance the insight into the workings of the program. In contrast, algorithm animation systems such as BALSA (Brown and Sedgewick 1983) provide a customised set of representations for the particular algorithm involved but usually require a specialist user to configure the system. Given the diversity of specifications that can arise, a general-purpose tool that can be used without specialist intervention would seem a much more useful facility.

4.4 Initial Requirements

The purpose of this section is to show how the desire to investigate the use of visualisation techniques in the formal specification of ADTs results in a set of high-level requirements that the intended system should satisfy. A rationale is presented for each requirement with appropriate discussion. These requirements are the results of lessons learned from the visualisation systems discussed in chapter three, as well as developing ideas about what a visualisation system for formally specified ADTs should be like. The design of VISAGE was made more difficult (and more interesting) by the absence of other systems for visualising formal specifications of ADTs that could act as a starting point for further development. VISAGE is therefore a first-generation prototype whose principal contribution will be to identify fruitful avenues whilst warning of dead-ends. The design

description will therefore include details of what went wrong as well as successful decisions.

4.4.1 Representing an Abstract Data Type

An ADT is an object without any explicit physical form and so to represent it in a computer requires mapping it onto a more "physical" representation. The most common mapping is onto a textual notation of words and symbols, for example, as an algebraic specification. As Smith (1977, p.37) notes, symbolic representations, such as used in mathematics, are attractive because they have many well-understood properties which can be manipulated independently of their referents, and once transformed, can then be mapped back onto their referents. For example, textual substitution in an ADT's formal specification can be used to rename the operations associated with that ADT.

As argued in chapter three, the best way to visualise a complex, abstract object is with several different representations, or views, each showing a separate facet of the object's structure or behaviour. ADTs are complex objects, having several different facets including a name, relationships with other ADTs, a set of associated operations, and so on. To display these different attributes in an understandable way will require several different representations since a single, all-encompassing representation would be incomprehensible to the user. VISAGE should therefore use multiple representations of an ADT to allow each of its facets to be considered in isolation from the others yet allow the overall representation to be grasped from the sum of the views.

4.4.2 Auto-Didactic Interface

Since most people have received over a decade of training to develop text-based reading and writing skills it is usually easier for them to use text as a way of handling formal descriptions. Pictorial notations require drafting skills that generally have not been developed to the same extent as for text and are consequently harder to use with the same degree of ease and precision. However, with the support of interactive computer graphics, pictorial descriptions could be just as easy and convenient to manipulate, and with the incorporation of suitable constraint mechanisms, graphics can have the same degree of precision, and hence formality, as text. Unfortunately, interactive computer graphics still requires powerful computers and although graphical workstations are becoming increasingly common and cheaper, it is unrealistic to expect that users will have such facilities to hand when specifying ADTs in the "real world", at least in the immediate future. Specifications are generally developed semi-formally using paper and pencil, only being entered into the computer for checking and other processing when they are nearing completion.

Given the dominance of textual notations, it is essential that people are trained in the skills of specifying software using these notations, i.e. they will have to be fluent in using textual specification languages. It is therefore not satisfactory to provide only pictorial representations of ADTs: a practical teaching system would also provide a way for novices to become familiar with textual notations, subconsciously developing intuitions and skills in using them. The "user friendliness" of graphics would support novices until they feel confident about using a textual representation on its own. The system effectively allows the user to transfer knowledge from one domain to the other.

The most common, precise textual representation of an ADT is its formal specification as illustrated by the examples developed in chapter two. For greatest effectiveness, the textual representation used in VISAGE should be a standard specification language. However, since there is as yet no standard syntax for algebraic specifications, an ad hoc syntax can be used in VISAGE so long as it is sufficiently similar to the dialects found in other (textual) specification systems. Whatever the concrete syntax used, the representation should be enhanced by the use of different fonts, text styles and sizes, following the techniques described in Baecker and Marcus (1986).

Consider a system with two synchronised views juxtaposed, one graphical the other textual, viewing an ADT. The user is allowed to manipulate the ADT through either view with any changes being automatically reflected in the other view. Users with little experience of algebraic specifications of ADTs will at first probably not understand the textual view of an algebraically-specified ADT with all its stylistic conventions and syntactic embellishments. However, as they manipulate the underlying ADT through the graphical view, the textual view is updated in synchrony allowing them to see the result of their action expressed in a textual format. Conversely, a user experienced in the formal specification of ADTs could acquire an understanding of the graphical view by performing actions in the text view. This kind of interface that teaches the user in a subliminal manner is termed *auto-didactic*. With experience, users will appreciate the benefits of each type of view developing intuitions and skill for using them for the tasks for which they are most appropriate. For example, textual representations are good for tasks such as global renaming, whereas graphical representations are good for structural editing.

4.5 Implementation Environment

A principal factor accounting for the surge in development of visualisation systems is the ready availability of powerful computer workstations equipped with high-resolution bit-mapped displays capable of showing high-quality graphics images as well as text. The VISAGE prototype was designed from the outset to run on such configuration, in particular, a Sun-3/140 workstation equipped with a monochrome display and eight

megabytes of primary memory. User input is by means of a keyboard and a three-button optical mouse. The host operating system for this machine is UNIX.

4.6 Choice of Implementation Language

The choice of implementation language is important because of the way it influences later design decisions. A language imposes a *mind-set:* a philosophy and methodology for tackling problems. Certain design decisions may require that a certain class of language be used, thus restricting later design options, but it is imperative that these be minimised.

4.6.1 Requirements

Given the desire to provide a graphical interface to an object, it is obvious that the language should either support graphics explicitly or could be linked to some suitably high-level graphics library. Support for graphical interaction with the display is also essential since the system is to be highly interactive. The need to support multiple views of an object implies the need for a window management system that provides direct support for the synchronised display of these views. From a software development point of view it is important for the language to be part of an integrated support environment equipped with powerful tools to assist in constructing software. Even these basic requirements reduce the available options to only a few configurations.

4.6.2 Options

Since algebraic specifications of ADTs are the objects of interest it seems sensible to extend an existing system that manipulates them, such as OBJ (discussed in chapter two), to provide a graphical interface as well as a textual one. Unfortunately, portability and availability problems excluded them from further consideration.

The next possibility was to use a standard programming language such as C or Pascal combined with a graphics library and window manager. Despite the (archaic) development tools available under UNIX (the standard operating system for workstations), this option would have required the most work to get a working prototype because of the primitiveness of these languages and the inflexibility of the monolithic graphics libraries. This option was dropped for these reasons.

Existing research systems dedicated to visual programming such as GARDEN (Reiss, 1987), were not considered for two reasons. Firstly, they were still experimental with restricted availability, and secondly, their adherence to traditional programming languages and methodologies did not make them suitable as a platform for ADT specification without considerable modification which greatly diminished their attractiveness.

The final option was to use a language with explicit support for graphics and the extensive interaction required by the prototype system. When the VISAGE prototype was originally being designed (late 1987), the one language that stood out in this category was Smalltalk-80. This language and its accompanying integrated programming environment, are the result of over a decade's research at Xerox's Palo Alto Research Center into a new computing philosophy, inspiring the vogue for Object-Oriented Programming (OOP). The development of Smalltalk was driven by the desire to give people their own powerful computer equipped with a highly-interactive graphical interface. As Cox notes:

> *"Much of today's enthusiasm for high-resolution graphics workstations, iconic user interfaces, and personal computing can be traced directly to the demonstration of the value and feasibility of these ideas [in Smalltalk]."*
>
> Cox (1986, p.30)

From this background alone Smalltalk seemed the natural choice. However, when the full power of the language and environment were appreciated, the reasons for choosing Smalltalk became overwhelming.

4.6.3 The Smalltalk-80 Language and Environment

The Smalltalk system does not readily decompose into what traditional environments call the programming support tools, language and run-time support system. Smalltalk is the archetypal object-oriented programming system being composed *entirely* from a set of *objects*. The uniformity of its design gives the system power, elegance and flexibility. The Smalltalk-80 language is described in detail in Goldberg and Robson (1983), with Goldberg (1984) describing the programming environment.

Objects consist of some private data and a set of operations (called *methods*): their similarity to ADTs should be apparent. Computation is done by sending a *message* to an object asking it to perform one of its methods. A method can modify the internal state of the object, can send messages to other objects and can optionally return an object as the result of the computation to the sender of the original message. A *class* describes the implementation of a set of objects that all represent the same kind of component. Each member of the set is an *instance* of the class. A class can have an arbitrary number of *subclasses* whose instances are the same as those of the parent class (called its *superclass*) except for explicitly stated differences. A subclass is said to *inherit* the properties (methods and internal data) of its superclass and is free to extend or modify these properties (without affecting the superclass). Since a subclass can itself have subclasses, the entire system becomes a tree-like inheritance hierarchy. Smalltalk-80 does not allow a class to have more than one superclass, i.e. it does not support multiple inheritance.

As an example of classes and inheritance consider an application that uses complex numbers: a class not supplied as standard. A new class Complex is added as a subclass of the Number class, refining the behaviour of inherited methods such as addition, and extending the set of methods to include operations such as returning the imaginary part of the complex number. Unlike programs written using traditional languages such as Pascal, Smalltalk programs are not neatly delimited. User programs are spread throughout the system thereby *evolving* additional functionality in an organic fashion, Brooks (1987, p.18). Although novice Smalltalk programmers find this holistic programming style difficult to grasp at first, it soon becomes completely natural.

Problems with Smalltalk

Although the Smalltalk-80 environment is based on relatively few concepts, it is a large system due to the comprehensive collection of tools and pre-defined classes made available to the programmer. The sheer size of the system makes it difficult for a novice user to form a conceptual model of the system's architecture with the result that the learning curve is initially quite steep. Although the standard Smalltalk texts† give detailed descriptions of the system, their coverage is not complete and the user is left to learn the system by using it and through computing folklore. This form of learning is possible in Smalltalk because powerful facilities are available to help a user understand what is happening, e.g. the *explain* command when applied to any fragment of source code will return a description of what that fragment does and where to find more information.

A further problem in learning Smalltalk is the novelty of programming in an object-oriented style. Programmers coming from a traditional imperative programming background will have to unlearn cherished concepts and learn how to use efficiently the machinery of object-oriented programming.

Smalltalk also has a number of technical problems. The most significant is the lack of static type checking, i.e. the ability to detect attempts at sending messages to objects of the wrong "type" at compile time. For example, the following code fragment is accepted by the Smalltalk compiler but which would cause a run-time error since the object pointed at by the var variable (an instance of class Set) does not know about the addition method.

```
| var result |
var := Set new.
result := var + 3.
```

Although such errors are trapped at run-time and can then be corrected using Smalltalk's sophisticated debugging facilities, it would be much better to have detected the problem at

---

† The standard texts are Goldberg (1984), and Goldberg and Robson (1983).

compile time. Some experimental Smalltalk developments, such as Johnson (1986), have tried to add a type-checking mechanism in order to provide strong typing but these are neither standard nor widely available.

To support a large, highly interactive environment with reasonable performance, Smalltalk demands large amounts of computer memory and processing power, commonly in the form of dedicated, powerful workstations with around eight megabytes of memory and a fast processor. Even with this support, the overhead of handling dynamic binding and method look-up means that Smalltalk programs never run as fast as languages close to the machine-level, such as C. This demand for high-performance hardware means that Smalltalk is still an exotic beast, although plummeting hardware costs should make it more readily available. This hunger for hardware stems from the belief that computers are cheap compared with the time of the programmer, and therefore the programming environment should provide support for as much of the process as possible.

4.6.4 Implications

What has the choice of Smalltalk as the implementation language given us, and what restrictions has it placed on future design options? Smalltalk has a large library of classes organised into its inheritance hierarchy. These include, among many others, support for general programming, graphics, interaction with the user through the mouse and menus, and an experimental framework for developing multiple view applications called the Model-View-Controller framework. The generality of these classes means that new applications can often simply reuse code directly. However if the pre-defined classes do not have the required functionality then the inheritance and subclassing mechanisms mean that simple extensions of existing classes are often all that is needed. The evolutionary approach to program development means that design experiments can be performed quickly with low risk since we can backtrack away from unfruitful avenues.

Smalltalk is unusual in allowing *all* the source code used in the system to be inspected and even modified. This includes code for "system" components such as the compiler and window manager. Although this openness may be dangerous for novices, it does allow expert programmers to tailor the system to their own preferences. Smalltalk's openness in allowing changes to be made to *any* part of the system means that design decisions are not constrained by the inflexibility of the language as would be the case in using languages and libraries of the traditional variety. The availability of tools such as browsers and change managers within the programming environment means that finding, reusing or modifying code is safe and easy. Smalltalk is therefore particularly good for the rapid prototyping of new software applications, especially those with highly interactive, graphical interfaces. Recent advances in OOP technology, such as efficient garbage collection mechanisms and

the caching of compiled methods† as well as more powerful computers has meant that Smalltalk can also be considered as a vehicle for the development of production quality software.

4.7 The Model–View–Controller Framework

It is important when concurrently displaying multiple views of an object that the different views are *synchronised.* This ensures that if the object being viewed changes in some way then *all* the views are updated to reflect the new state of the object with no view left displaying stale information. Smalltalk-80 has a built-in mechanism for handling this synchronisation between an object and its views called the *Model–View–Controller framework*, usually shortened to MVC. This framework is an experiment in constructing highly interactive, graphical user interfaces, and as Smalltalk's designers admit, it has its limitations. To compound matters, the MVC is *not* described in the available official texts, leaving developers the daunting task of understanding its workings through folklore (e.g. Cunningham, 1985) and browsing the source code.

The architecture of the MVC framework is shown in figure 4.1. This architecture partitions a program into two major parts: the underlying application is the Model; the user interface is handled by the combination of View and Controller.

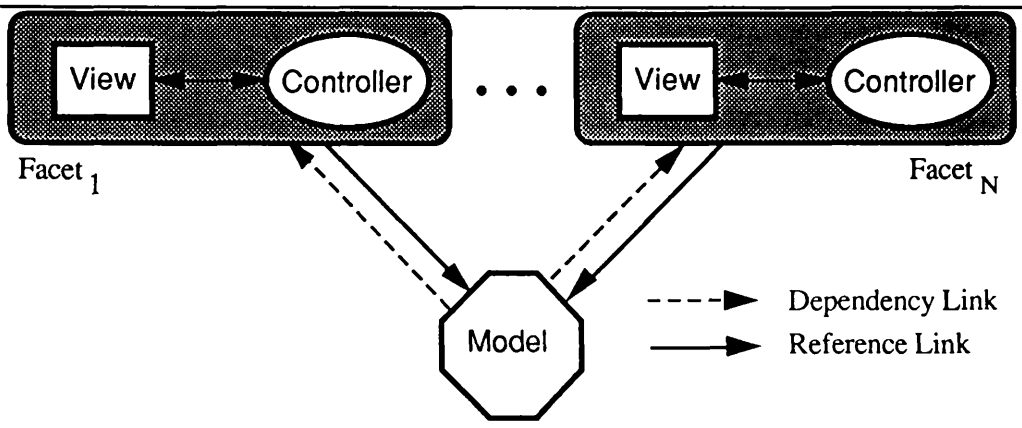


Figure 4.1: Architecture of the Model-View-Controller Framework

The *Model* can be thought of as an abstract object that encapsulates some data and has a number of operations that it can perform. A *View* displays a particular facet of the model: it is responsible for the output aspect of the interface. The *Controller* interprets user inputs in its associated view and, depending upon the action, may ask the model to perform a

† Although Goldberg and Robson (1983) define the semantics of Smalltalk in terms of an interpreter operational model, recent implementations of the language compile Smalltalk into native machine code.

certain operation. Once the model has changed, it informs all dependent views so that their display can be updated if necessary.

To illustrate this architecture, consider the following simple example. A variable, acting as the Model, is storing an integer number representing a temperature constrained to be in the range zero to one hundred Centigrade. Three separate View–Controller pairs are showing different facets of this model:

1. a slider gives a graphical indication of the value of the Model. Its Controller module lets the user drag the slider using the mouse thereby changing the value of the number held by the Model;

2. a coloured box gives a pictorial representation of the current temperature with 0°C shown as blue and 100°C as red with a spectrum of colours in between. This view's controller lets the user alter the temperature by clicking on mouse buttons;

3. the last view shows the temperature as a numeric display with the controller letting the user edit the number to set the temperature explicitly.

As shown in figure 4.1, the components of the triad are linked together using two types of linkage. The reference link allows one component to refer to another, to obtain information or ask it to perform a particular operation. For example, the View can ask the Model for information it needs to produce the display. Note, however, that the Model cannot refer to either the View or Controller. The dependency link shown in figure 4.1 establishes a dependency between two components. In this case the View is dependent upon its Model: it is informed of any changes in the Model so that it can update the display, if necessary, to keep the Model and display states consistent.

The power of the MVC derives from the facility to have more than one View–Controller pair associated with a particular Model as shown diagrammatically in figure 4.1. Each view is a dependent of the model and will be informed of any changes to the model. Since each view is informed of changes, this mechanism ensures that all the views are synchronised (at least within the limitations of a sequential processor). In an ideal world, these views would be updated in parallel after any change, however the current implementation of Smalltalk treats each view update sequentially, operating on each view in a round-robin manner. It requires considerable fine-tuning to arrange that the dependent views are updated in the most pleasing order. One attempt at removing this awkwardness by having the views updated in parallel is discussed in Altmann et al. (1988).

Although the MVC architecture partitions a system into interface and application aspects, this simple conceptual framework complicates the construction of interactive systems. The developer has to decide how to partition functionality and responsibility for different parts of the interface between the view and the controller. The lack of an explicit programming

methodology results in applications that lack structure and modularity. Since expertise in using the MVC is gained through study of existing applications and folklore, it takes considerable time to master the subtleties of developing such applications.

A second major problem with the MVC concerns the mechanism used for informing all dependent views that a model has changed in some way. The dependency links between a model and its views are simply tokens sent by the model to each view. Each view examines the token to see whether it knows how to deal with this particular change: a non-trivial computation. This mechanism is very expensive if there are many views (as in VISAGE) and only one view that will respond to a particular token. The problem is compounded by the need in highly interactive interfaces for a plethora of tokens to handle the many different changes that are possible. It would be much more efficient if change announcements could be directed to only those views that need to be updated.

The MVC framework has the further problem that the dependency link "glue" is smeared throughout the application. Since any member of the MVC can ask the model to announce a change, trying to understand an application's behaviour involves unravelling this web of dependency relationships. Uncharacteristically, the Smalltalk environment fails to provide support in handling this web which makes the task of developing interactive applications extremely error-prone.

4.8 Foundations of VISAGE

Before describing the design of the more novel aspects of VISAGE, it is necessary to look at the infrastructure that supports these higher levels. These foundations include the management of the display area, the protocol for selecting objects and issuing commands, hold information is stored, etc. These aspects are discussed in the following sub-sections.

4.8.1 Tiling versus Overlapping Views

Given the desire to display the different facets of an ADT using multiple textual and graphical views, how should these views be arranged on the limited available screen area of a workstation? To discuss the options requires fixing some terms of reference: the terminology of Smalltalk will be used. In Smalltalk, a *window* is a separate display entity that can be individually collapsed to an icon, expanded from its icon, resized, moved and closed (i.e. discarded). Many windows can be displayed on the screen at the same time, with the potential to overlap or even totally obscure each other. Each window can contain one or more *views* with the constraint that views within a single window cannot overlap or obscure each other. These points are illustrated in figure 4.2.

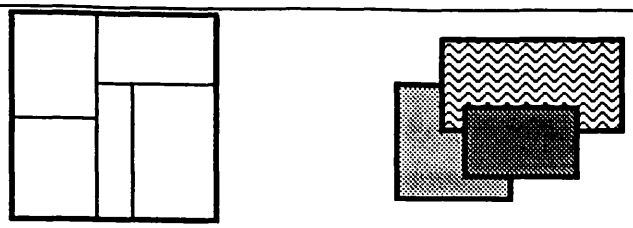


Figure 4.2: Tiling of Views and Overlapping Windows

Given that VISAGE is composed of several different views related in various ways, how should the views be arranged so as to make the presentation of information as clear and helpful as possible? The options form a spectrum from having each view in a separate window to having all the views within one window. In choosing an arrangement it is necessary to understand the trade-offs. Having a view in its own window means that it can be independently manipulated. The user is able to customise the windows according to personal preference and the task at hand with the overhead of having to manage a plethora of windows. Having all views in one window allows the user to see the overall situation but at the expense of not being able to adjust individual views and having cramped displays. In between these extremes are the various combinations.

In an empirical comparison of overlapping and tiled arrangements, Bly and Rosenberg (1986) report that a tiled arrangement is better for users who do not have skills in managing windows, e.g. bringing windows to the front, or moving and resizing them. Since it is likely that VISAGE users are in this category it seems sensible to arrange the views in a tiled layout within one window. However, it is not necessary to display all VISAGE views at the same time. The various views may be partitioned into functionally-related groups that could be naturally placed within their own window. This decomposition seems to offer the best compromise in terms of giving greatest user control whilst presenting related views together. This approach was adopted in VISAGE where the views for autonomous tools such as the PLAYPEN (described in chapter six) were grouped into their own window.

4.8.2 The Mouse and its Menus

Even the most minimal application involves a certain degree of user interaction. With VISAGE, the user must, at least, select a specification to visualise and perhaps scroll views to reveal information that is not currently within the window. In addition, the user may want to perform certain actions on displayed objects, e.g. ask for a screen dump of a graphical view, or obtain help information. To handle such user interaction requires

careful design if the system is to be pleasant and easy to use for a wide range of users, and novices in particular.

The Smalltalk system allows the user to interact with an application using the keyboard and a three-button mouse as the pointing device. The mouse is used to specify locations on the screen and for issuing commands held on pop-up menus. The Smalltalk system expects a three-button mouse and assigns each button a particular generic task. For example, the left button is always used to specify a location on the screen, or to select displayed objects. Pressing the middle mouse button causes a context-sensitive pop-up menu to appear with commands appropriate to the view currently containing the cursor. Pressing the right button causes a different pop-up menu to appear, this time with commands for manipulating the entire window e.g. moving, closing, collapsing and resizing. This menu appears irrespective of which view the cursor is in, so long as it is within the window boundary. The following sub-sections discuss the mechanisms for selecting objects and using the menus.

Selecting Objects

As described above, the left mouse button is used for selecting the object at the current cursor location. Objects can be either textual or graphical depending upon the particular view. In VISAGE, selectable objects can be one of the following: elements of a list, text strings, buttons, icons, and links. It is essential that a consistent approach is used for selecting all of these objects so as to minimise what the user must remember.

The basic procedure for selecting an object has two parts: the user first positions the cursor over the object and then clicks (press then release) the left mouse button. Since the selection is only made when the button is released, moving the cursor away from the object before releasing the button will cancel the selection. The visual effect of selecting an object depends on its type, e.g. selecting an icon will cause it to become highlighted (e.g. by displaying its label in reverse video); selecting a position in a text string places a text insertion caret (see figure 4.4(i)) at that point ready for text entry from the keyboard. The effect on the system of a selection is also dependent upon the object, and the view where the selection was made.

Command Menus

Commands in the VISAGE system are issued by selecting an option from a menu of alternatives. As described at the start of this section, pop-up menus can appear when the user presses either the middle or right mouse buttons. This mechanism was chosen because of its advantages when compared to the alternatives of a fixed menu and a pull-down menu. Firstly, a pop-up menu only appears when the mouse button is pressed, making efficient use of screen space. Secondly, it appears at the current cursor location which avoids the problem of the user having to make time-consuming and distracting eye

and mouse movements to find and select the desired command. Lastly, it is relatively straightforward to make the menu sensitive to the current state of the system, thereby ensuring that only valid command options are presented for selection.

The procedure for selecting a menu option follows a *noun–verb* style regardless of whether it is the middle or right mouse button that is pressed. This style has the advantage of avoiding the modes that occur with verb–noun selection styles. Figure 4.3 shows the steps involved: menus are shown in the upper half of parts (b) and (c); the bottom parts shows the state of the mouse buttons with a black rectangle representing a button depression. The user the subject of the command by placing the cursor over the object (the "temp" icon shown in figures (a) and (d)). When the button is pressed the menu appears at the cursor location (figure (b)). While keeping the button depressed, moving the cursor over the elements in the menu will cause the one under the cursor to be highlighted: figure (c) shows the third element highlighted. This feedback is used to indicate which command will be executed if the user released the mouse button. When the button is released the menu disappears and the currently highlighted option (if any) will be executed (figure (d)).

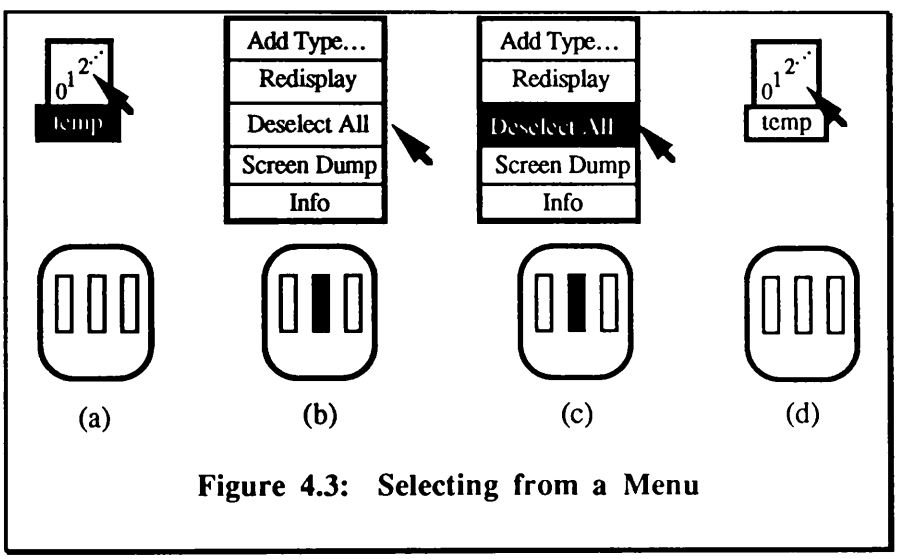


Figure 4.3: Selecting from a Menu

If the user has already selected an option from a particular menu then the next time that menu appears that option will be automatically highlighted. This has the benefit that if a command is being issued several times in succession then selections can be made very rapidly. However, this feature can confuse users: it is discussed in greater detail in chapter seven which presents an analysis of an evaluation of VISAGE.

4.8.3 The Storage Medium

Textual notations have evolved over thousands of years as a concise and efficient way of storing and transmitting information, and it seems counter-productive to forgo these advantages, even in a highly graphical system. In particular, computer memory technology and networks have been designed to handle text characters as the basic unit of

information. To store pictorial information on a computer requires special encoding techniques before it can live in this textual universe, and then takes up relatively large amounts of storage when compared to text even when special encoding techniques are used. For example, an A4 page of text on the Macintosh computer requires about four kilobytes of disk storage whereas the storage requirements for a picture of the same dimensions are one or even two magnitudes greater, depending on the complexity of the picture. Given this difference in storage requirements it seems obvious that a textual format should be used as the representation for storing ADTs on a long-term basis.

This storage policy demands that information supplied graphically by the user can be converted automatically into an equivalent textual representation. This has the considerable advantage that ADT specifications can now be used outwith VISAGE, for example, as input to other tools. This aspect is considered in the *Future Work* section of chapter eight.

4.8.4 Screen Dumps

Although VISAGE stores ADT specifications in a textual format, there are times when the user would like a permanent, graphical record of some facet of an ADT. These pictures could be used as documentation or to show to other people, e.g. in teaching. To this end, VISAGE allows the user to create screen dumps of every graphical view by issuing a menu command. The screen dump differs from the displayed view in having a white background instead of the usual gray as this was found to make clearer hard copies due to the improved contrast between background and objects displayed in the foreground. After issuing the command the user is asked whether the file should be saved in Smalltalk Form or MacPaint format. The former is useful if the picture is to be further processed within the Smalltalk environment; the latter is useful for transferring to a Macintosh computer for editing or incorporation into documents or overhead slides. In generating the screen dump in MacPaint format, VISAGE automatically scales the picture to ensure it fits within the resolution limitations. A screen dump produces the smallest rectangular image that includes all the view's contents, even those parts that are not visible in the window.

4.8.5 Logging of User Actions

As VISAGE is intended as an introductory environment for novices, it will be important to know how the system is being used so that a user's problems can be identified at an early stage. The information required will consist of details of the commands being issued, what views the user favours, how many error situations arise and how long the user spends on certain tasks. The best way to gather these data is by having the system monitor the user's activity automatically. Since the amount of data is potentially very large, a useful option would be to specify what aspects of the user's behaviour are of particular

interest, e.g. the error rate in using the mouse. It would also be useful if the system could produce a report of a session that summarises the user's activity.

The VISAGE prototype includes the facility to monitor user activity to help identify any problems they may be having in using the system. This facility can be enabled or disabled when the system starts executing. The monitor time-stamps and records details of the following events: commands selected from menus, object selections, user and system errors. When VISAGE quits, a summary of this log is printed to a file for later analysis.

An extension of this facility would record how long the user spends in each view, using the assumption that the cursor location is the focus of their attention. This would give a reasonable indication of the usefulness of each view for particular tasks, and especially in just how well graphical and textual views complement each other. An improved mechanism would allow all user actions to be recorded for future play-back and analysis. This feature was omitted from the VISAGE prototype because of the computational overhead in monitoring user events, particularly mouse events, with acceptable frequency.

4.8.6 Displaying State Information

Despite the increasing processing power of computers, there are tasks that require a significant time to perform, e.g. accessing a remote device or copying a large file. In addition to these busy states, VISAGE has several other transient states, for example, waiting for the user to select an entry from a menu. It is imperative that the user be made aware of this state to prevent them being concerned at lack of system response, or to indicate what the system is expecting from the user. Similarly, whenever the user issues a command, the system should indicate receipt of it to reassure the user that it is now being dealt with, so that the user does not try to issue the command again. This feedback is achieved by means of several devices discussed in the following sub-sections.

Journal

A standard way of indicating what the system is currently doing is to have a status box for messages describing what is happening. Usually these boxes allow only single line messages that are overwritten by the next message. Often important messages flash past the user, lost forever. The VISAGE browser includes a status area called the *Journal* that allows time-stamped messages to be posted for the user's information. The Journal differs from the usual approach in that the user can scroll the view back to look at previous messages, thereby ensuring that important information is not lost.

In early versions of the VISAGE browser the Journal was used as the place to post error messages, e.g. announcing that a specification file contained a syntax error. However, these messages were often overlooked since the user's attention was typically not on the Journal. For this reason a separate error reporting mechanism was developed: this is

described in a later section. The Journal is now used for indicating which files have been read and written by the system, and other non-critical status information.

Cursor Shapes

A simple and effective way of indicating a particular system state is to change the shape and image of the cursor, with different states given a different cursor. The rationale behind this approach is that the cursor is usually the focus of the user's attention and so any change in its shape or appearance is likely to be noticed. VISAGE uses some of the cursors provided within the standard Smalltalk system to indicate its states; these are shown in figure 4.4.

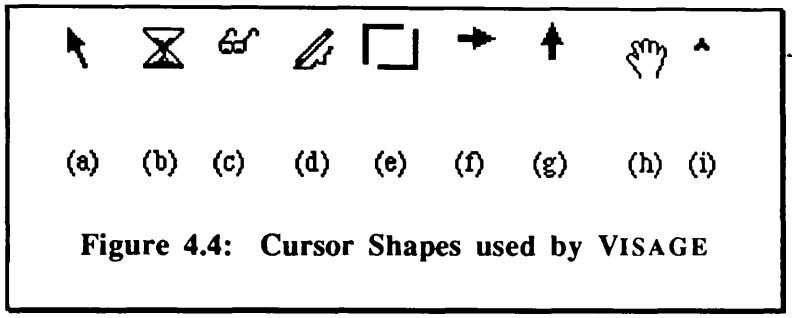


Figure 4.4: Cursor Shapes used by VISAGE

Cursor (a) indicates that VISAGE is waiting for the user to select an object or issue a command; (b) appears when executing a lengthy command; (c) means the system is reading a specification file; (d) denotes that VISAGE is writing a specification to backing store; (e) is a pair of cursors that appear when the user is asked to size and position a window on the display; (f) and (g) are used in scrolling the view in the Y and X-axes respectively; (h) allows the user to "grab" the view and scroll it in two dimensions at the same time; (i) shows the insertion point within a text view i.e. where characters typed at the keyboard will appear.

The Window Label

Smalltalk windows usually have a small textual label above their top-left corner that identifies the application. The application window can be collapsed (to conserve screen space, for example) leaving only this label on the display. The window can be restored simply by double clicking on the collapsed label. Usually the contents of this label are fixed to be the name of the application. The VISAGE browser extends the use of this label to include overall status indication as well as application identification. In particular, the label indicates if the user has edited a specification using the browser but has not saved those changes to a file.

Progress Indication

Although the use of different cursor shapes gives a useful indication of system state, its static nature gives no clue to how quickly the task is progressing. If the system can also

give a regularly updated indication of the task's progress then user anxiety will be reduced. There have been several attempts at providing such an indication, either at the operating system level or within a single application. One approach displays a clock-face with a single seconds hand that continuously sweeps across the face. Such an indicator only shows that the task is progressing with no clue as to when it will finish. A better approach would calibrate the clock sweep so that one revolution corresponded to the life-time of the task, e.g. if the hand starts at twelve then the six o'clock position means the task is half finished. By noting the rate of sweep and the position of the hand, the user can estimate when the task will finish. Empirical studies described in Myers (1985) show that users prefer systems that have progress indicators.

VISAGE provides a calibrated progress indicator to supplement the cursor shapes in indicating state and progress information. This indicator is used for lengthy operations such as transferring specifications to or from backing store, and updating the different views after a change in an object being displayed. A problem with the calibrated sort of indicator is that the system must compute how long the processing of a particular task will take. With tasks such as reading a file the system can calibrate the indicator by the length of the file. However, with some tasks the system cannot compute beforehand how long the task will take.

Consider the situation of updating the display after a change in the Browser. One change usually gives rise to several separate broadcast messages (call this number n) that have to be sent to m different views. Since Smalltalk handles view updates in a sequential manner, this one change is broken down into `n * m` individual updates. By calibrating the progress indicator into `n * m` steps and advancing it after each atomic update, VISAGE gives an acceptable (albeit coarse) progress indication. The reason for the coarseness is due to differences in the times to process different atomic updates.

Attention Seekers

In user-driven interfaces, the user is free to operate on objects and issue commands in any order they choose. However, in certain rare situations the user must perform a certain action before proceeding. An example of such a situation is where the user tries to over-write an existing file. A safety-conscious interface (such as VISAGE's) will demand confirmation before proceeding. With VISAGE a special dialogue box appears explaining the situation; the user must then press one of two buttons that say whether the operation should proceed or not. Until the user has pressed a button, the system will block all other input and will cause the box to flash if the user tries to ignore it, e.g. by moving the cursor outside the box's boundary. This mechanism is used in VISAGE for dangerous operations that must be dealt with such as deletion, quitting without saving changes to specifications, and requesting names for displayed objects.

4.8.7 Handling Error Situations

When novices use a system they will inevitably make mistakes, by issuing inappropriate commands, for example. In many cases it is possible to avoid certain error situations through careful design of the interface, e.g. by disabling commands that are illegal at a given time. However, it is not possible to avoid all error situations and in such circumstances, the system should detect and catch any error the user (or system) makes. Since users may not have a lot of experience in using computers, it is essential that recovery from error situations should be viewed positively without making the user feel helpless or inadequate, Shneiderman (1982).

Error messages should not accuse the user of making an error but simply explain the problem, suggest a remedy and allow the user to continue, if appropriate. The message should also explain how the user can get more detailed information if this message still leaves any doubt about what to do next. Although such a message requires more support from the system, it simplifies the user's task of fixing the problem quickly and correctly, and results in a more flexible and pleasant system.

4.8.8 Reducing Screen Clutter

A common problem with visualisation systems, as described in chapter three, is the tendency for the size of representations to grow quickly with increasing complexity of the object being displayed. Many systems have suffered from their attempt to squeeze a lot of graphical information into an overcrowded display area. Although the graphical representations used to visualise ADTs have been designed so as to minimise their greed for display area, there will be occasions when a representation will not fit within the area of the screen reserved for the particular view. There are several solutions to this problem and these are discussed separately below. In describing the options it is convenient to think of the complete representation as being drawn on some arbitrarily large canvas with the display acting as a window onto this canvas. This situation is shown in figure 4.5.

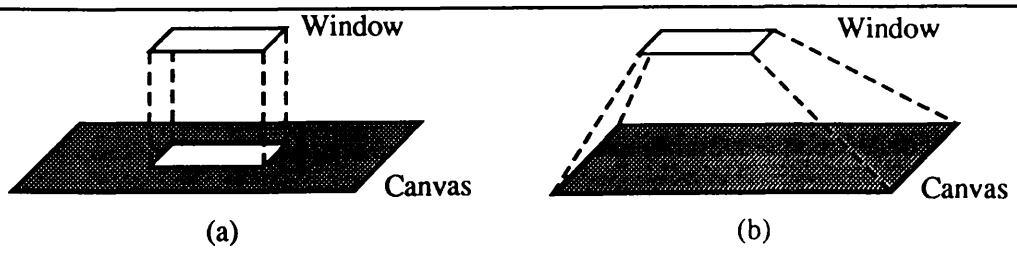


Figure 4.5: Two Ways of Mapping a Large Canvas onto a Small Window

Scrolling

One of the most straightforward approaches is to show only a part of the canvas with the remainder being clipped at the view boundary, as shown in figure 4.5(a). The user can see the parts that are off-screen by scrolling the window to the desired area of the canvas. One-dimensional scrolling mechanisms are common in text editors, allowing the user to move back and forth in a file. Such a mechanism was generalised to handle the two-dimensional canvases used in VISAGE. Horizontal and vertical scroll bars appear whenever the cursor is moved into the view's area. The scroll bars show the relative position and size of the window compared with the overall canvas. Scrolling along either axis is achieved by dragging the scroll bar's "elevators" using the mouse.

The scroll-bar mechanism restricts scrolling to only one axis at a time: this quickly becomes irksome. To overcome this restriction, a more direct mechanism was implemented that allows the view to be scrolled in both axes simultaneously. Whenever the user presses the left mouse button when over the background of the view, the cursor "grabs" the canvas and drags it around tracking movements of the mouse. The scroll bars are updated as the canvas is dragged about. The cursor changes to an image of a hand (shown in figure 4.4(h)) while the mouse button is pressed. The directness and naturalness of this mechanism made it the preferred scrolling style for all users of VISAGE observed in the evaluation (c.f. chapter seven).

Zooming

An alternative approach to scrolling maps the entire canvas onto the window with suitable scaling to make it fit, with corresponding loss of detail: this is shown in figure 4.5(b). Detail can be restored by letting the user zoom in (preferably with varying levels of magnification) on a particular area of the canvas. This approach suffers from the problem that at high magnification the user can forget his position within the overall canvas, with the potential of misunderstanding the view. Some systems (the PV system (Brown et al. 1985), for example) overcome this by presenting two views: one showing the close-up view and the other showing the overall picture, highlighting the magnified region. This split-screen approach was dropped as it would be too expensive in terms of display space.

The *Fisheye* approach described in Furnas (1986) manages to combine the advantages of magnified and overall views within one display, giving local detail and global context together. The window onto the canvas can now be thought of as a lens whose power of magnification varies over its surface, with the user fixing the strengths at each point. Although this mechanism was not implemented due to practical reasons, it would be an interesting future experiment to compare it with the scrolling mechanism that was adopted.

4.9 Developing a Graphical Vocabulary

Visualising an ADT requires mapping its attributes and structure onto a graphical representation. Chapter two regarded an ADT specification as being composed of the following components: the set of operations associated with the ADT, the behaviour of these operations, the relationships this ADT has with other ADTs, and the set of values the ADT can have. In order to map these aspects onto graphical attributes it necessary to first consider what options are available, i.e. decide upon a graphical vocabulary that can be used as the basis of a language for describing ADTs in a graphical manner.

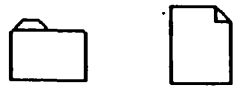
A workstation display is capable of showing arbitrary images using a matrix of small picture elements (or *pixels)* each of which can be individually made black or white†. To produce a useful display requires imposing structure on this raw painting facility. A common way of doing this is to draw closed geometric shapes such as rectangles or circles in order to split the display into areas, i.e. the parts inside and outside the shape's boundary. Such an image is perceived by the brain as an distinct entity and can be used to represent some object. However, if the number of objects to be represented becomes large, the set of different geometrical shapes required quickly becomes unmanageable with the user unable to remember what each shape denotes. Rather than give each object a unique graphical shape, an easier approach is to use an identification scheme based on classifying objects according to some similarity. As an analogy, consider the set of all Scotsmen, each identified by the tartan of his kilt. Clearly, giving each Scot his own tartan would produce an unmanageable (but colourful) set of identifiers. However, by assigning a tartan to a related clan of Scots results in a smaller range of tartans which is easier to remember. With this approach a secondary identifier is required to distinguish the members within a clan, thus Hamish who wears the MacKay tartan. In this analogy the MacKay clan acts as a sort with its family members representing the values that belong to the sort. Just as with clans, so objects of different types can be represented by a single shape, with objects of the same type distinguished by a secondary identifier, such as a label.

This two-tier naming approach is attractive for use in graphical user interfaces as the user has to remember only a limited set of basic graphical representations. This approach has culminated in the computer-oriented iconography popularised by the user interface of the Xerox STAR computer (Smith et al. 1983) and later the Macintosh computer where a small set of usually rectangular shapes (called *icons)* represent abstract entities such as files and folders. Two simple examples from the Macintosh user interface are shown below (Williams 1984).

---

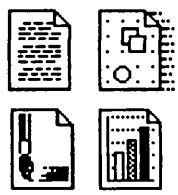
† The extension of this display model to include colour is discussed in chapter eight.

The left-hand icon represents a folder, corresponding to a directory in conventional operating systems such as UNIX; the right-hand icon represents an (unclassified) file. The shapes of these icons were chosen (after considerable testing) so as to resemble the equivalent objects to be found in an office environment, thereby simplifying the task of learning what the graphical symbols mean. This is particularly important if the interface is to be easy to use by a wide variety of people.



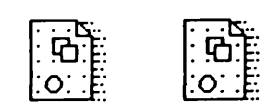
One problem with the file icon shown above is that it gives no indication as to the contents of a file. A file might, for example, contain a spreadsheet, an executable program, or a chapter of a thesis. It would be useful if these different types of file could be represented within the basic iconic framework. Typically, this is done by giving the icon a characteristic image with files containing the same sort of information being represented by the same icon.

Some examples from the Macintosh interface of icon images being used to indicate file contents are shown on the right. Starting at top-left and working clockwise, these icons represent the following classes of file: a text file, a drawing, a histogram, and a free-hand sketch.



This iconography is still not capable of distinguishing between two different entities of the same type, for example two files would be shown as two instances of the same icon: which icon represents which file? There are a couple of ways to overcome this problem. Icons can be distinguished by placing each in a unique location on the display. For example, users of systems with "desktop" interfaces (such as the Macintosh) commonly place frequently used application programs or files at easily accessible and memorable locations on the screen. The use of such "spatial identifiers" can greatly improve the ease and efficiency of accessing commonly-used entities and has been incorporated as a design principle in many user interfaces. Although this may work for a small number of entities in the user's "working set", the load on the user's memory becomes too great when the number of entities grows beyond a small number (Miller, 1956). To avoid this it is common to assign to each icon a unique, non-spatial identifier to describe the file's contents, e.g. a textual label .

An example of distinguishing between icons of the same type as done in the Macintosh interface is shown on the right. The two icons have the same, system-supplied image as they both represent files containing graphical images. The icons are distinguished by their labels. These textual identifiers are chosen by the user to describe the contents of the files. Although they are meaningful to the user, they are of no use to the underlying file manager.



The iconography of the Macintosh user interface provides a limited vocabulary for representing the different entities to be found in a document-oriented environment. Although, very successful and popular, the interface does not exploit some of the other graphical constructs that are available, e.g. an obvious idea is to use different sizes of icon to denote different sizes of file, thereby giving the user an idea of how long it would take to process a particular file, Cockshott (1989).

Objects in the world are usually only interesting when they are related to other objects. When these objects are represented in a user interface it is essential that all the relevant relationships between these objects are given a graphical representation. For example, in the Macintosh computer, files are held in folders, and folders are stored on disks. These relationships are given a graphical representation based on physical containment on the display. When a folder or disk is *opened* to reveal its contents, the files or sub-folders it contains are displayed within the physical boundary of the folder's image. This organisation has important benefits. All the objects with the same parent are kept in the same logical space. A closed folder acts as a holophrast that stands for the set of objects it contains, allowing them to be treated as a unit, i.e. it abstracts away the details of its internal structure.

The use of physical containment to denote a *belongs-to* relationship can be exploited in visualising ADTs as a way of grouping together the operations associated with a particular ADT: the ADT would be represented by a (holophrast) icon that when opened would reveal the operations. However, since containment is only suitable when there is a strict hierarchical relationship between objects, it is not appropriate for representing the more general relationships found with ADTs. For example, as figure 2.3 shows, a directed graph is required to represent the relationships between operations and their argument and resultant sorts, and this cannot be naturally coerced into a containment format. The standard way of graphically representing arbitrary relationships between objects is through *node-and-arc* diagrams, where the nodes represent the objects and the (optionally directed) arcs relate the objects it connects in some specified way. Such diagrams have a long history and strong theoretical basis in *graph theory*. By using different line styles for the arcs, these diagrams can show many different relationships simultaneously. However, too many different types of arc will almost certainly confuse the reader. The main drawback with these diagrams is that the user must be told what the relationship each type of arc represents, by means of a key, for example. By careful design, these diagrams can convey information about relationships in a clear and insightful manner.

4.9.1 VISAGE Icons

VISAGE uses small pictures (called icons) to represent ADTs and their operations. In an early version of the prototype the icons for these two classes of entity shared the same anatomy which is shown in figure 4.6(a) and (b).

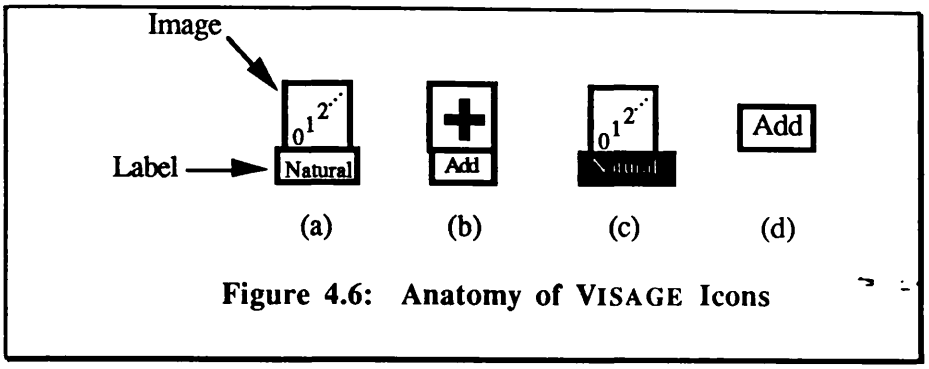


Figure 4.6: Anatomy of VISAGE Icons

An icon has two parts: a user-supplied image and a label showing the name of the object. The label can be highlighted using reverse video to indicate when the icon is selected. An example of a highlighted icon is shown in figure 4.6(c). The icon's image is intended to convey some intuitive meaning to the user about what this object represents. VISAGE provides a default image for all newly created icons and the user then tailors it to something more meaningful using Smalltalk's (primitive) icon editor.

This simple approach suffered from two problems. Firstly, although it is relatively easy to devise images for ADTs (nouns) it was difficult to come up with satisfactory images for operations (verbs). The basis for this is that we are used to seeing static images of entities in the world but not of actions. Actions are dynamic processes that can only be given a static image with difficulty: a rare example is shown in figure 4.6(b). The second problem was that many icons were being generated and having the user edit all of them (especially given the poor editing facility) was too onerous. For these reasons it was decided to change the representation for operations and dispense with the image part. The revised form is shown in figure 4.6(d): the icon is just a label made up from the operation's name.

If the user goes to considerable effort to design and draw an image for an icon the least the system can do is remember the drawing for use in later sessions. Every time the user edits an image it is marked as needing to be saved and then stored in backing store in a special icon directory when VISAGE quits. When VISAGE starts executing, all the images stored in this icon directory (including special images required by the system) are read into a Smalltalk dictionary that is accessible to all the graphical views.

4.9.2 Links between Icons

Having dealt with a representation for the nodes in a node-and-arc diagram it is now necessary to represent the arcs. A simple arc (as used in VISAGE) connecting two nodes denotes a relationship that exists between the objects represented by these nodes. Arcs can optionally be directed to show ordered relationships such as A *greater than* B.

The usual way of displaying arcs is as a line segment (a *link*) connecting the two node end-points. Directed arcs can be shown as a link with an arrowhead to indicate the direction of relationship. Just as a diagram may have several different sorts of nodes, so several different relationships may need to be displayed in the same diagram. This can be done using a primitive vocabulary of links that uses a variety of line styles (e.g. dashed or full), line widths, and arrowheads to represent the different relationships. Unfortunately, users find it difficult to discriminate between different line formats and user interface guide-lines usually recommend limiting line formats to a small (e.g. less than four), easily discernible set. VISAGE uses only combinations of thick and normal widths, and dashed and plain styles, giving four formats in all although only at most three are used in any one view. Despite using only a small number of distinct formats the user may still forget what they denote, particularly if the same format represents different relationships in different views. To overcome this problem the system should provide a key that describes the relationship. In VISAGE this is done using the *Help* facility to be described in a later section.

4.10 Summary

This chapter has considered the integration of the algebraic specification of ADTs within a graphical environment. This lead onto a discussion of the basic design requirements for a system that supports this novel association. A reasoned argument was presented for the use of Smalltalk-80 as the implementation language. This included a review of its limitations and the ramifications of choosing Smalltalk on the design of VISAGE.

This chapter also contained a description of the infrastructure needed to support a graphical specification environment including the development of an appropriate graphical vocabulary and interaction mechanisms.

The following chapter builds upon these foundations by adding a collection of textual and graphical views that represent the various facets of an ADT's structure and form.

Chapter Five
The VISAGE Views

5.1 Overview of the VISAGE Architecture

This chapter presents the design of the VISAGE system for visualising the algebraic specifications of abstract data types. The need for each of the main views is discussed, followed by a detailed account of its subsequent development. This account includes a look at the unsuccessful approaches that were investigated. The chapter also describes the VISAGE specification language and the mechanisms for supporting comments and help information.

Given the complexity of VISAGE it is difficult to grasp the internal structure of the system, and so as way of an overview, this section presents a high-level view of VISAGE's architecture shown schematically in figure 5.1.

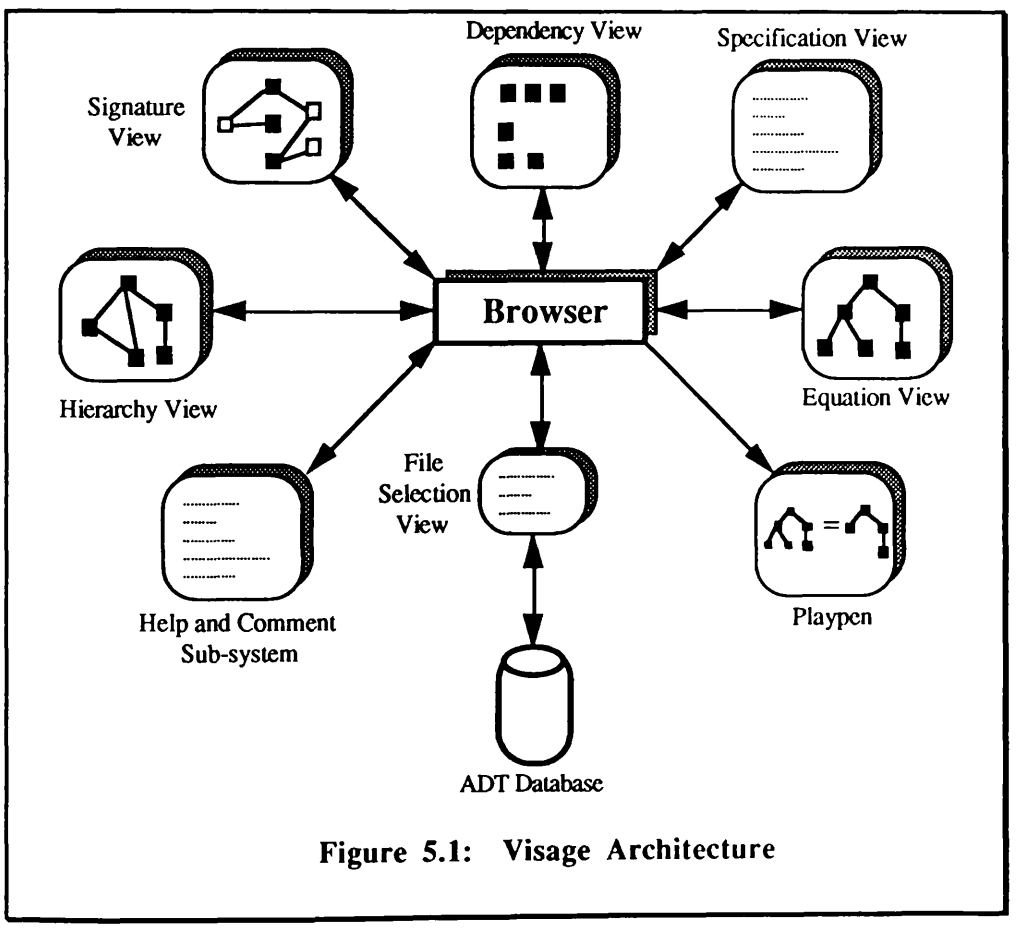


Figure 5.1: Visage Architecture

Central to VISAGE is the Browser which supports the visualisation and editing of ADT specifications. As well as acting as the main repository for state, configuration set-up, and ADT and icon data, the Browser handles the synchronisation of the various views

ensuring that displays remain consistent irrespective of which view the user is currently interacting with. The arrows in figure 5.1 show the directions of information flow around the system. The information includes specifications, icon images, selection updates and so on.

Once a particular file of specifications in the ADT database has been selected, its contents are parsed by the Browser to become a forest of abstract syntax trees, each representing a different ADT. The Browser is responsible for performing a variety of checks on these ADTs. The relationships between the ADTs are displayed in the Hierarchy View which also allows the user to select an individual ADT for more detailed examination. The selection of an ADT causes VISAGE to display the ADT's "internal" structure using separate views to show the different facets:

- the Specification View displays a pretty-printed textual representation,
- the Signature View shows its operations' signatures,
- the Dependency View shows the relationships between these operations,
- the Equation View displays dendritic representations of individual equations.

In addition, whenever an ADT is selected the PLAYPEN allows the user to "animate" it by having user-supplied terms rewritten using the ADT's equations.

The Help facility automatically generates context-sensitive help information for any displayed entity or view. For entities representing specification components (for example, operations or ADTs), the user can attach comments to augment the system-generated information.

5.2 Selecting Specification Files

When storing ADT specifications it seems sensible to locate those that are related in the same place, for example, in the same file. The specifications for different projects and systems would therefore be spread across several different files. In addition, libraries of commonly used specifications could be collected together into central files to act as a source of reusable components. Given this distribution of specifications, a browsing tool for specifications should allow the user to select a file that contains a specification he is interested in. A simple and error-free mechanism for selecting a file is to present the user with a menu of all the available specification files and have one selected simply by clicking on its name. Not only does this approach avoid the spelling errors that can arise if the user must type the filename, it simplifies the selection since the user has the much easier task of recognising a filename rather than remembering it. This textual menu can be thought of as a view of the file-store that filters out all files that do not contain specifications; it is called the *File Selection View.*

5.3 Specification View

Rather than just display an unformatted listing of the currently selected specification, VISAGE exploits Smalltalk's ability to display text in a variety of styles, fonts and sizes. This allows the different components of a specification to be distinguished in a visually obvious way. The chosen format shows comments in italic roman, keywords in large bold roman, and other components in roman style. When a signature or equation is selected it is emboldened; disabled entities are shown in small italic. These textual formats readily present selection status information while still allowing the user to see the entire specification. An example of specification formatted in this way is given in figure 5.2.

```
Datatype: string uses: char, natural;
    "Textual strings of characters."

    null: → string    "Create a new, empty string."
    concat: string × string → string    "Combine two strings."
    size?: string → natural    "How many characters in the string?"

Equations:
    size? (null) = zero
    size? (concat(s1, s2)) = add(size?(s1), size?(s2))
EndSpec
```

Figure 5.2: Example of a Formatted Specification

The pretty-printing mechanism takes the abstract syntax tree for the current specification and converts it into a formatted textual equivalent: it is the inverse of the parsing process. All the classes used to represent the various nodes in the syntax tree have a method to convert their part of the tree into a concrete textual representation complete with formatting instructions. The final result of the pretty-printer is a formatted text string that can passed onto an output stage. Usually the formatted specification is displayed in the Specification View. However, if the text is being sent to a file then the formatting instructions are converted to standard codes that are recognised by many terminals and display utilities such as *more* in UNIX that use effects such as reverse video to highlight text. In addition, a post-processor has been written to massage the file into a format suitable for a laser printer so as to produce high-quality listings of specifications.

5.3.1 The VISAGE Specification Language

As discussed in chapter two, algebraic specifications are written in a formal specification language that gives a precise meaning to statements written in that language, i.e. precisely defining the behaviour of an ADT. The original goal of the research described in this thesis was to take such a specification and make it more understandable through use of program visualisation techniques. An obvious requirement therefore is a language definition together with a parser that allows the system to process such specifications.

The specification language used by VISAGE was influenced by several different dialects and systems that have been presented in the literature. The language evolved throughout the research to cope with extra constructs, e.g. initially, only signatures were allowed, then simple equations, then conditional equations, comments and so on. The sub-sections below describe the various aspects of the language.

It should be borne in mind that the VISAGE specification language is only a means to an end and exploits only conventional techniques in language and parser design. Although a textual specification is required as the input to the visualisation process, it was felt unnecessary to devote excessive efforts to making this language particularly attractive and easy to use, since such efforts have been applied by other researchers. The language aims to be simple, yet capable of handling a wide variety of specifications. Cosmetic features such as mixfix syntax or operator overloading, for example, are therefore not provided in the language which considerably reduces its complexity (and hence that of the parser) without sacrificing descriptive ability. A complete BNF description of the language's syntax is given in Appendix B.

5.3.2 The Parser

The parser in the VISAGE browser takes the algebraic specification of an ADT and converts it into an internal representation more amenable to machine processing. The specification can come from two sources: either from a file containing a set of related specifications or from the Specification View which facilitates the direct editing of specifications. In either case, the parser is presented with an identical linear text string of the specification.

The parser is of the recursive-decent variety whereby the syntactic structure of the language is directly mapped onto calls to Smalltalk methods that parse particular constructs. This parser construction was chosen because it is simple to implement and executes efficiently on a system using a run-time stack (as is the case with Smalltalk).

Internal Representation

The output of the parser is an abstract syntax tree corresponding to the structure of the input specification. Each node in the tree is represented by a Smalltalk object of an

appropriate class. A portion of a typical syntax tree is shown in figure 5.3; the collapsed parts being represented by ellipses.

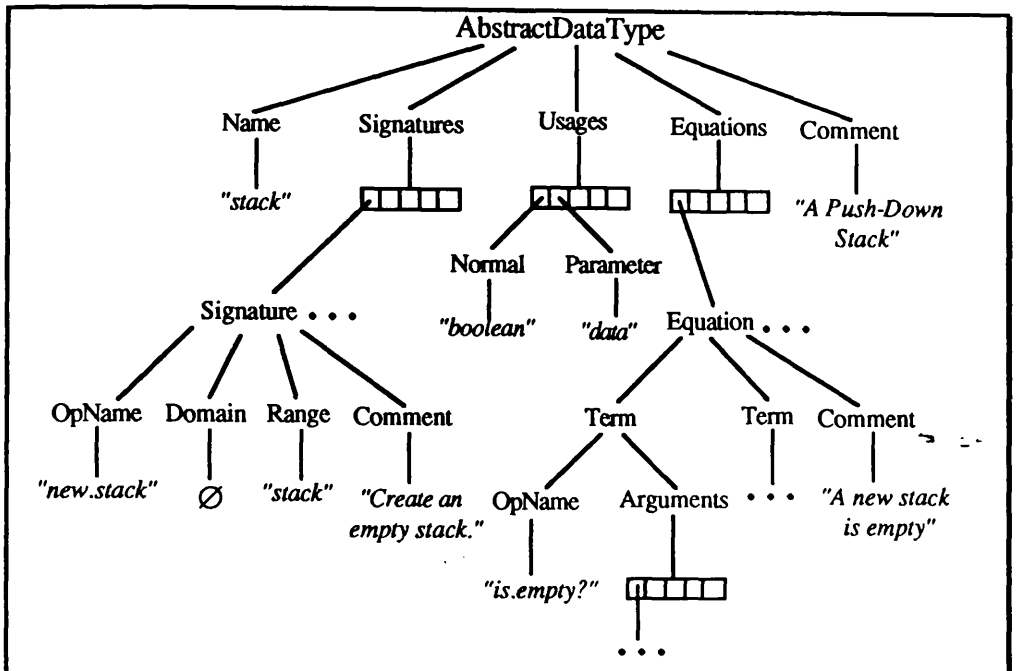


Figure 5.3: A Portion of an Abstract Syntax Tree

Commonly, related specifications are grouped together into a file so that they may be processed and considered as a unit. When VISAGE reads such a file it parses the individual specifications and places the resulting syntax tree in a dictionary object with the ADT name as the index. Since only one specification can be viewed at any one time, VISAGE records the current selection using the index name of its entry in the dictionary. A file of specifications when parsed becomes a forest of specification trees. VISAGE performs several checks on this forest to ensure consistency among the specifications. For example, if one specification uses another then that second specification must exist in the dictionary. In addition, there cannot be any dependency cycles i.e. A using B and B using A. Any discrepancies are reported to the user for optional repair.

Specification Errors

When a specification is edited textually within VISAGE, or when reading an existing specification from a file, there is the possibility that specification errors will be introduced. This is because the text editing facilities provided within VISAGE are context free, providing no constraint on what the user may type. A variety of different errors can result.

Lexical errors are the simplest sort and are usually the result of mis-typing a keyword, for example. Next are syntax errors caused by the specification not adhering to the grammar of the specification language, e.g. equations appearing in the signature part. Both of these

sorts of error could be avoided by the use of a syntax-directed editor. A third category of error is reference to unknown operations or ADTs, often due to using the wrong identifier. Lastly, the equations in a specification may give rise to type errors of various kinds when an object of one sort is expected but an object of a different sort is given instead. Alternatively, an operation may have too many or too few arguments.

When a specification is parsed any lexical and syntactic errors that are found are reported to the user. Parsing stops at the first error encountered. This simple approach avoids the complexity (and dubious value) of trying to repair the mistake automatically and then continuing. The reports take the form of a pop-up window containing a diagnostic error message together with an indication of where that error is likely to be found. An example of such an error is shown below. The italic text describes the error in encountered; the following line shows a fragment of the specification around the point where the error was detected (indicated by the arrow).

Missing Closing Parenthesis

```
size?(push(d, s) = succ (size?(…
                  ↑
```

The error report allows the user to locate the error in the specification quickly, and effect a repair. Since specifications will usually be textually edited using the Specification View it may seem sensible to insert the error diagnostic directly into the displayed text at the appropriate point (as done by the Smalltalk compiler). Although the user's attention is immediately focussed on the error, the user has the chore of having to edit out the error message as well as repairing the error. The VISAGE approach of having the error report in a separate window means that the specification is not tampered with by the system and the message remains on the screen as a reminder for as long as the user needs. Since the Smalltalk editor can be asked to find the occurrence of the context string using a single command, the advantage of inserting the error marker directly into the text is insignificant.

The Handling of Variables

The equations in a specifications can contain variables that are place-holders for terms of the appropriate sort. Since the specification language used by VISAGE does not allow overloading of operators, it is always possible to determine the sort of a variable simply from its context. When developing a specification (or program), it is common to use variables before declaring them, and then later supply the declarations. This style is error prone because programmers often omit some declarations. Some systems (such as Smalltalk) will automatically declare variables introduced by the programmer.

The approach adopted in VISAGE is to allow the writer of a specification to use variables as required *without* needing to insert an explicit declaration. The parser spots variables as it processes a specification, and by examining the context can determine the variable's sort.

These details are then added to the symbol table associated with each ADT. Once a variable has been implicitly declared, any subsequent instances are checked to ensure they are of the same sort as the original instance.

5.3.3 Pre-defined Abstract Data Types

When writing specifications it is common to build a new ADT on top of other, lower-level ones. At the base of such a structure are common building blocks such as the Boolean, Character and Integer ADTs. Rather than have the user specify these basic ADTs for every new specification they could be provided as part of the standard specification environment. There are two ways of doing this.

The first places them in a fixed library built into the system. With the second, a prelude file containing the specification of these ADTs is supplied to the system and processed immediately before processing a user specification. The former is used by languages such as PASCAL where the run-time library supports operations on the built-in types such as Integer. It has the disadvantage that the library cannot be altered or inspected. The second option has the advantage that the user can inspect, extend or even modify the prelude file according to their individual specification needs. The disadvantage is that the prelude file must be processed before processing of the user's specification can start which can cause a slight delay. However, the flexibility and openness of the second approach outweighed the disadvantages and so was adopted in VISAGE.

The user can provide any number of prelude files depending upon the application and can even override the system default file. When a new specification file is selected, all files with extension `.prelude` are read into the system. The file `standard.prelude` contains the specification of the Boolean and natural numbers ADTs. Since Boolean is needed for handling conditional equations, if its specification is not present then an error message is issued and processing stops.

5.4 VISAGE Help Facility

Novices, by definition, will not be fully conversant with the facilities available in a system and therefore need instruction into what is available and how to use it, as well as information about what things mean and do. The provision of rich and comprehensive help facilities is considered to be an essential requirement for a system intended to be used by novices. Naturally, conventional printed help material for VISAGE is available to the user. This includes a tutorial introductory guide and a reference document that gives a brief summary of VISAGE commands and facilities. However, printed material is not the complete solution to providing help to the user. Often, manuals will not be at hand when the user encounters a problem; even with an index, it may still be difficult to locate help

information on a particular topic. A powerful and convenient complement to printed help material is an on-line help facility that is sensitive to the current system state and context.

A context-sensitive on-line help facility has been implemented within VISAGE. At any time the user can request help information about any displayed entity by issuing the omnipresent info command that appears on every menu. The result of each request for help information appears in a separate Smalltalk window, allowing the user to consult the information and interact with the VISAGE system simultaneously. When the information is no longer needed, the window can simply be closed. The contents of the help window are displayed using different fonts, character sizes and styles to highlight and demarcate important information. The help information is not a fixed description but is instead *generated* by VISAGE when needed so as to take into account the current state of the system and user activity. This mechanism ensures the help information is more perspicuous and flexible than typical approaches where help information is fixed or hard-coded within an application.

With a large, complex system like VISAGE, it is impractical to present the user with a long description of every system feature and facility, and have him search for the section appropriate to the current context. Instead, the user should be supplied with only pertinent information allowing them to overcome their difficulty and resume the task at hand. VISAGE determines what information to generate by looking at the current cursor context i.e. what was the user pointing at when the help command was issued. There are three basic categories of entity each generating a different type of help information:

Icons Describes what this icon represents, for example, an operation or ADT, together with specific details such as where it was defined and how to select it to obtain more information. A summary of the menu commands applicable to this icon is included.

Links Describes what relationship this link represents and indicates its connectivity i.e. what entities are at its end-points. A summary of the menu commands applicable to this link is included.

Views Describes the purpose of the view giving a description of the different objects that can appear, including what the objects represent and the variety of relationships that exist between them. Details are given of how to move around the view, select objects and the set of commands available within the view.

5.5 Comments in Specifications

Even with a flexible, context-sensitive help facility such as that provided by the VISAGE help facility, the information presented still has a system rather than user perspective. The

user often wants to record informal notes or comments about what a particular entity is for and how it should behave. Most programming languages allow such comments to be added to the source code of programs. Although these comments are ignored by the compiler they provide informal intuitions that can help the user understand the workings of the program. For identical reasons, the textual specification language used by VISAGE for describing ADTs allows informal comments to be added to the specification.

Batch-oriented compilers for languages such as Pascal treat comments as noise that must filtered out before translation can begin. The syntactic definition of such languages often omit comments for sake of simplicity. With the arrival of programming environments, comments must be treated with more respect. A program (or specification) is now held as an abstract syntax tree with comments associated with a particular node in the tree. This means that if the tree is restructured, e.g. by rearranging declarations, then the comments will be moved correspondingly. In contrast, if the declarations in a Pascal program are moved about, the comments remain where they were, relying on the programmer to move them. Requiring that comments are associated with a particular node in the syntax tree limits the freedom for placing comments in a program. Comments now appear explicitly in syntactic definitions associating them with a particular syntactic entity. For example, the BNF description of the VISAGE language given in Appendix B includes the definition:

SIGNATURE ::= IDENTIFIER : FUNCTIONALITY [COMMENT]

```
push: data × stack → stack  "This builds stacks."
```

The optional comment here is associated with the `push` signature node in the syntax tree. VISAGE has the convention that comments follow the syntactic entity with which they are associated. The user can supply a new comment or edit an existing one by typing in the Specification View and issuing the ACCEPT command.

It was soon realised that the VISAGE help facility was most commonly used in conjunction with the comments shown in the textual Specification View. A natural development was therefore to try and combine these two sources of information. For example, if the user requested help information on, say, an operation it would be useful to display the comment associated with that operation at the same time. To this end, the VISAGE help facility was extended so that if the user requests information on any object capable of having a comment associated with it, then the help window would have two views, as shown in figure 5.4.

The top view is read-only and contains the system-generated help information for the selected object. The lower pane contains the comment associated with that object. This lower view can be edited by the user to add or modify a comment, and when accepted, this message is incorporated within the syntax tree of the specification. Modifying a comment causes the Specification View to be updated to reflect the change.

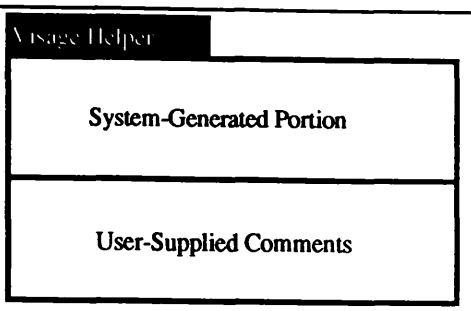


Figure 5.4: Window with Combined Help and Comments

5.6 General Problems with the Graphical Views

Considered at their most abstract, the objects being visualised in VISAGE are graph or tree structures. Unfortunately, the different facets of an ADT's behaviour or structure demand hand-crafted representations in order to provide suitably perspicuous views that appeal to both intuitive understanding and accepted conventions, precluding the use of a single graph drawing utility. Although the views are specialised they share common, general problems regarding their generation.

Considerable theoretical and practical attention has been given to the optimal aesthetic layout of arbitrary graph structures: a problem known to be NP-complete (Johnson 1982). The problem becomes practicable only when the demand for an optimal solution is relaxed: as the quality of the final output decreases, so the ability to compute the layout in acceptable time and space increases. A successful display algorithm must therefore find a suitable compromise between aesthetic appeal and the computational investment required. Since it is recognised that the user will want to customise and rearrange the display no matter how good the computed layout, the system can trade-off speed of generating a display against its aesthetic quality. A poor display can be produced relatively quickly compared with a high-quality one, but will require more effort on the user's part to customise the display to his liking.

Many of the visualisation systems discussed in chapter three failed to gain acceptance by users because they were too slow in producing and updating the display of even moderately sized objects. To avoid this problem with VISAGE, it is essential that the user is not kept waiting for too long before the display reflects the new state of the system. Obviously, the qualifier "too long" is open to subjective definition but can be taken to be of the order of a second for highly interactive applications. This can be partially achieved in a brute-force way by using powerful workstations but a more elegant solution requires careful design of the display algorithms to ensure that they are as efficient as possible.

5.7 Hierarchy View

An ADT specification does not exist in a vacuum but is usually a member of a family of specifications with various dependencies between them. An obvious requirement is the clear representation of these family ties. Usually, these relationships are not manifest in a textual specification, leaving the reader to unravel dependencies by following chains of implicit references. This often leads to problems due to the user changing a specification without knowing all the ramifications of his action and perhaps thereby introducing inconsistency into the overall specification. By making the relationships between specifications explicit i.e. by displaying them in a suitable way, this class of problem can be avoided. Since computer graphics is particularly good at showing relationships between objects, it seems sensible to represent the relationships between ADTs using a graphical view. This requirement gave rise to the Hierarchy View: when the user views a file containing a group of specifications the Hierarchy View produces a graphical display of the relationships that exist between them.

Since a set of related ADTs could be very large, it would not be practicable to view them all at the same time. Typically, a user is interested in the behaviour of one ADT, and although its relationships to other ADTs is important, it is not of primary concern. Therefore, a mechanism should be provided to let the user designate which ADT is of interest. This could best be done by having the user select the ADT's graphical image in the Hierarchy View.

The original intention in the Hierarchy View was to show only usage dependencies between ADTs but, as will be discussed later, the view can naturally express other, more exotic, relationships. In developing a view of a set of related ADTs it seemed natural to arrange the display in a top-down fashion: if ADT A uses ADT B then A appears above B in the display. In this way, ADTs at the bottom of the display are at the "lowest level", i.e. they are not dependent upon any other ADT. Note that it is perfectly valid for the graph to be disconnected: this would correspond to separate families of specifications saved to the same file. When a specification file is read in by VISAGE a directed graph of usage dependencies is created. The nodes of the graph are ADTs with edges between them representing the dependencies: an edge from node A to node B means that ADT A uses ADT B. It is this graph that is displayed in the Hierarchy View.

5.7.1 The Layout Algorithm

The Hierarchy View simplifies the layout of the usage graph by placing the graph's nodes onto a regular rectangular grid of points with row one at the top and column one on the left of the grid. The placement algorithm ensures that only one node appears at any given grid point.

A node's position on the rectangular grid is computed in two stages: one for the rows and then one for the columns. If an ADT is not being used by any other ADT then it appears on row one; otherwise, it appears one row lower than the lowest row of all those ADTs that use it. This can be described more formally as:

$$row(N) = 1, \text{ if } users(N) = \{\}$$
$$row(N) = 1 + \text{MAX}\ \{row(M) \mid M \in users(N)\}, \text{ otherwise.}$$

where $users(N)$ is the set of all ADTs that use the ADT N,
and $row(N)$ is the row number assigned to the ADT N.

This is effectively a topological sorting of the graph to produce a list of node sets where each set contains all the nodes to appear on the same row. The next step of the placement algorithm spreads the nodes of each row along the columns. By starting with the top row and considering entire *family trees*, the algorithm tries to place connected nodes so as to minimise the lengths of edges between nodes and the number of edge crossings in the graph. Once the placement of the graph's nodes has been completed, the final stage is to render the graph on the display. The ADT icons are placed at their computed grid locations with directed arcs drawn between connected nodes. Although the placement heuristic is fast and relatively successful, there is no guarantee that it will find the optimal layout for a given graph. This is regarded as acceptable given the design decision to let the user manually re-arrange the graph layout.

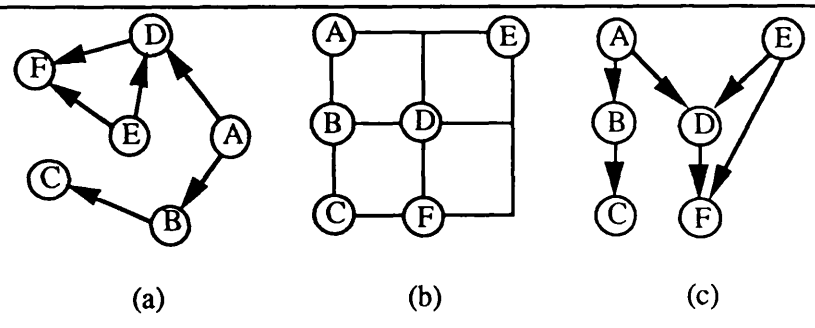


Figure 5.5: The Main Stages of the Hierarchy Placement Algorithm

Figure 5.5 illustrates the main stages of the Hierarchy View's placement algorithm. Figure (a) shows the usage dependency graph supplied as input to the view. Table 5.1 shows the application of the above algorithm to this graph.

ADT N	users(N)	row(N)
A	{ }	1
B	{ A }	1+ MAX { 1 } = 2
C	{ B }	1+ MAX { 2 } = 3
D	{ A, E }	1+ MAX { 1, 1 } = 2
E	{ }	1
F	{ D, E }	1+ MAX { 2, 1 } = 3

Table 5.1: Assignment of Icons to Rows in the Hierarchy View

After the topological sort into rows as shown in the table 5.1, the nodes are arranged among the columns of the grid using a bottom-up algorithm. A record is kept of the next free column in each row to ensure that no two icons are every placed at the same row and column. The following algorithm is then applied to each leaf node in the hierarchy graph (i.e. a node that uses no-one else). The leaf node is assigned to the next free column for its row. Then, in a breadth-first recursive manner, assign those nodes that use this one (its ancestors) to unique columns, regardless of which row they are on. If a node has already been assigned a column then do not move it unless it would violate the principle of ancestors being in unique columns.

Step Number	Nodes Being Considered	Algorithm Action
1	Leaf node C	C placed in column 1 of row 3.
2	Ancestors of C = { B }	B placed in column 1 of row 2.
3	Ancestors of B = { A }	A placed in column 1 of row 1.
4	Leaf node F	F placed in column 2 of row 3.
5	Ancestors of F = { D, E }	D placed in column 2 of row 2. E placed in column 3 of row 1.
6	Ancestors of D = { A, E }	A and E already in unique columns.

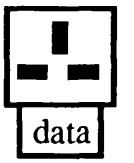
Table 5.2: Assignment of Icons to Columns in the Hierarchy View

To illustrate the column placement algorithm, consider its application to the example shown in figure 5.5. The leaves of this graph are nodes C and F as determined from table 5.1. The algorithm starts by placing node C in column 1. C's sole ancestor B is then placed in column 1 as is B's ancestor A. This completes the sub-graph reached from leaf C. The algorithm then moves onto the next leaf node, F, placing it in column 2 (since

column 1 in this row is occupied with node C). F has two ancestors, D and E, that must be placed in unique columns: D is placed in column 2 and E in column 3. The algorithm then considers the ancestors of D, namely A and E. Since both of these have already been assigned unique columns they are not moved. This completes the assignment of nodes to columns. Table 5.2 summarises these steps with figure 5.5(b) showing the placement of nodes in their final rows and columns. Figure 5.5(c) shows the final result (somewhat simplified) as generated by this layout algorithm. An example of layout in the Hierarchy View's display is shown in figure 5.6.

5.7.2 Extending the Basic Hierarchy View

The basic Hierarchy View described above shows ADTs as nodes connected by edges representing usage dependencies. When later extensions to the specification language allowed parameterised ADT specifications, it seemed desirable to visualise these new relationships between ADTs. To recap, a parameterised type uses a formal type parameter to stand for a range of actual types, for example in the stack[data] ADT, data is the formal type parameter. The relationship between stack and its data type parameter is important and should be visualised. Since the basic Hierarchy View already displays relationships between ADTs, it seemed natural to extend the view to handle these new relationships.



Although type parameters are place-holders for actual ADTs they are not proper ADTs and so need to be visually distinguished from ADTs in the Hierarchy View. The chosen approach uses the special iconic form shown opposite to represent a type parameter. The icon's image was made to resemble an electric socket to promote the metaphor of plugging actual types into the parameter. The label of the icon shows the name of the type parameter. In addition, a way of distinguishing the new parameter relationship from the ordinary usage dependency is needed. Since relationships are represented in the Hierarchy View by directed arcs, it seemed simple and natural to represent different relationships by different styles of lines. Accordingly, a dashed line is used to connect an ADT icon with its parameter icon.

Figure 5.6 shows an example of the output from the Hierarchy View that includes the extensions just described. It is the visualisation of a file that contains the following ADT specification headers:

```
Datatype: stack [data] uses: Boolean, natural;
Datatype: xref uses: string, stack;
Datatype: string uses: char;
Datatype: natural uses: Boolean;  "Primitive"
Datatype: Boolean;  "Primitive"
Datatype: char;  "Primitive"
```

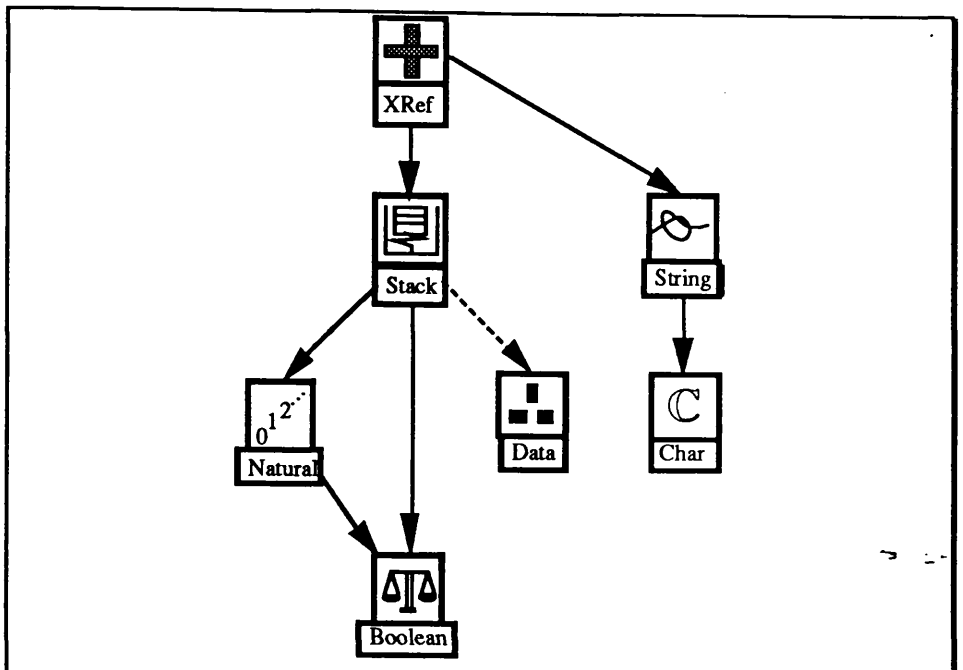


Figure 5.6: An Example Layout in the Hierarchy View

5.8 Signature View

The syntactic part of an ADT specification can be represented by a graphical representation called a *Signature Diagram*, an example of which is shown in figure 5.7. An advantage of a signature diagram over the more usual textual notation is a healthy reduction in the amount of syntactic sugar; compare the differences in figure 5.7.

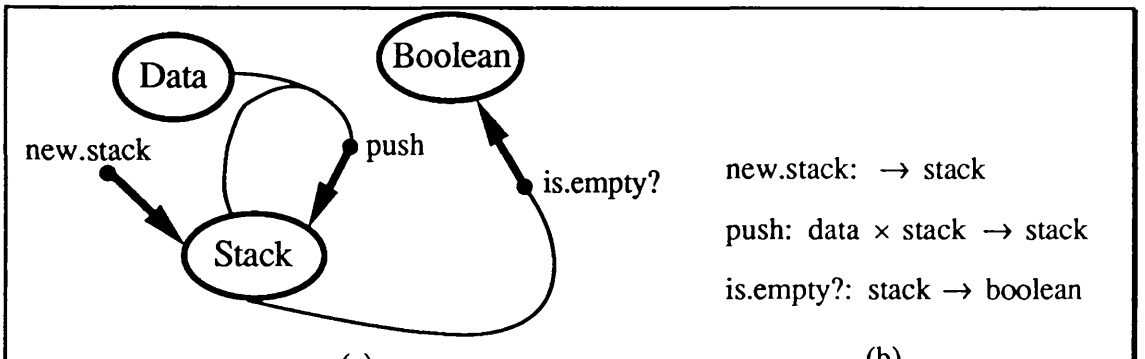


Figure 5.7: Text and Graphic Signatures

The use of diagrams to describe the syntax of a language is not new: the syntax of Pascal is often described using so-called Railroad diagrams, a kind of recursive transition network. Since signature diagrams are probably the only "standard" graphical representation commonly used by the algebraic specification community, it seems sensible to

adopt the format in VISAGE as the way of representing the signature of an ADT. It will be referred to hereafter as the *Signature View.*

In figure 5.7(b), the different components of each signature are separated by arbitrary characters that the user must learn: these are replaced in the graphical case by directed arcs as shown in figure 5.7(a). This graphical convention appeals to an intuitive dataflow, pipeline model of how operations work: input values flow into the operation with the single result flowing out; the direction of flow is indicated by the arrows. The graphical representation has other advantages. By combining the signatures of all the operations, the diagram shows, in a compact manner, how the operations are related as far as sorts are concerned. By following the different arcs, the user can see how terms can be constructed as well as their resultant sort. In addition, the textual signature does not readily distinguish between operators and sorts, representing both using simple identifiers. In the diagram, operators and sorts are given different graphical images (a dot and ellipse, respectively), which makes it much easier for the user to tell them apart by simple inspection.

As discussed in chapter two, traditional signature diagrams, such as shown above, suffer from some deficiencies. For example, the diagram does not give an indication of the order of the operands, e.g. is the data operand the first or second argument to the `push` operation? In addition, the operations are not given a very prominent graphical form, being represented by a simple dot.

Despite their limitations, signature diagrams have been incorporated within at least one other specification system: the ASSPEGIQUE system developed by Bidoit and Choppy (1985). Their implementation generates diagrams similar to the one shown in figure 5.7(a), even faithfully reproducing its deficiencies with no attempt to improve the diagram by using more sophisticated graphical techniques.

To overcome the deficiencies of the signature diagram, while retaining its simplicity and clarity, the VISAGE Signature View was developed. This view introduced a number of extensions to the basic representation; these are summarised below with fuller discussion following in later sub-sections:

- the view automatically labels the input arcs for operations with more than one argument parameter, to indicate the order of the operands;
- the operations are given a more prominent graphical form, being represented by an icon instead of just a small dot;
- partial operations (i.e. those not defined over their entire domain) are now explicitly represented;

- operations with the same signature, e.g. add and subtract over integers, are given their own graphic representation and are not collapsed together. The need for this is discussed in chapter six when considering the visual programming of signatures.

An example of the display generated by the Signature View that includes examples of extensions one and two above is shown in figure 5.8 and should be compared with the original in figure 5.7. Extension three is illustrated in figure 5.10 to follow.

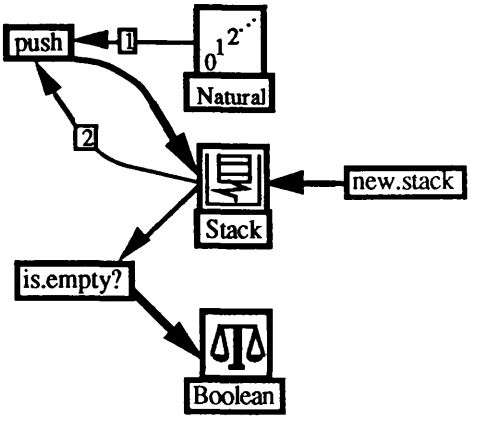


Figure 5.8: An Example of a Signature View Display

5.8.1 The Layout Algorithm

Given the signatures for the operations introduced by an ADT, the Signature View automatically generates an enhanced signature diagram of the form shown in figure 5.8. Since the diagram is an acyclic, directed graph, it suffers from the same general problems for layout as the Hierarchy View discussed in the previous section. Unfortunately, because it does not have the same hierarchical structure, the layout algorithm of the Hierarchy View could not be used: a specialised layout algorithm had to be developed. This section describes this development of this algorithm.

A signature diagram is a bi-partite graph whose nodes represent operations and sorts, with edges relating operations to their argument and result sorts. The layout algorithm takes as input three sets: a set of icons representing operations, a set of icons representing sorts, and a set of edges between operation icons and sort icons. The layout strategy is based on the observation that in general a particular sort will be connected (by virtue of it being the argument or result sort) to more than one operation. Therefore, the icon used to represent that sort should be placed in a location easily accessible to all the operations that are connected to it so as to minimise arc lengths and number of arc crossings.

The first layout algorithm based on this observation took an iterative approach to the placement of sort icons. If there is only one sort then clearly the optimal layout is to place the sort icon in the middle of a circle with the operation icons arranged equidistantly

around the circumference. With this arrangement, the operation icons are equidistant from the sort icon with no crossings of arcs. In the general case, the diagram is structured as two concentric circles: the sort icons being arranged around the inner one, the operation icons around the outer. Unfortunately, as more sort icons are considered, the number of arcs crossing the inner circle increases dramatically, making the diagram difficult to understand. The situation is illustrated in figure 5.9(a).

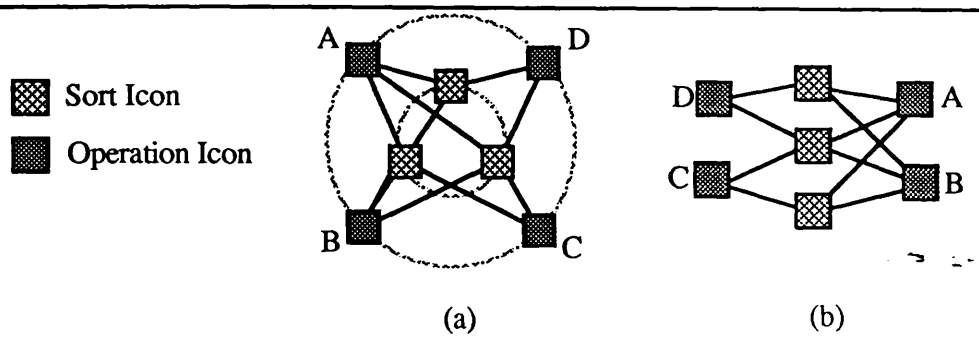


Figure 5.9: Alternative Signature View Layouts

Clearly, arranging the sort icons in an inner circle is not successful. However, the basic idea of placing operation icons near the sort icons they use still offers some promise of a good representation. Therefore, experiments were made with a variety of placement strategies, the most successful being the one shown in figure 5.9(b). Here the sort icons are arranged along a central line with the operation icons arranged on lines either side. With judicious placement of the icons, the basic arrangement is considerably improved.

The sort icons are placed first. By looking at the signatures, the algorithm determines which sort icons should be grouped together. Typically, certain sorts often appear together in several signatures and it makes sense to group them together as this will reduce the length of the arcs to the corresponding operation icons. For example, in the specification of the stack ADT (see Appendix C), the sorts data and stack are often used in the same signature and would be placed together by this algorithm. Note also that since the stack sort appears in every signature it makes sense to place it in a central location since it must be accessed by arcs from every operation icon.

Once the sort icons are in place the operation icons are arranged about them. The layout algorithm minimises arc length by placing each icon as near as possible to its *favourite* sort icon. For example, the `push` operation with signature `data x stack → stack` has stack as its favourite sort with data as second choice. The algorithm computes the various degrees of favouritism shown by the operations for different sorts and arranges the icons accordingly.

5.8.2 Relating Operations and their Sorts

Once both the sort and operation icons have been positioned, the directed arcs between them are added to complete the view. In the traditional signature diagram, the signatures were represented using named, multi-tailed arrows with the tails of the arrows attached to the argument sorts and the arrow-head terminating at the resultant sort (c.f. figure 5.7). The Signature View splits each multi-tailed arrow into two separate arrows for the head and tails, joined by an icon representing the operation (c.f. figure 5.8).

In keeping with the tradition introduced with signature diagrams, the result arc leaving the operation icon is shown thicker than input links. This makes it easier to identify the resultant sorts of operations. The problem of distinguishing multiple input arcs is overcome by automatically labelling the arc with a number indicating the position of this argument in the operation's argument list.

5.8.3 Handling Partial Operations

With many data types there will be certain operations associated with that type that are not defined for all the type's values. Such operations are termed *partial*. An example of a partial operation for the stack ADT is `top` which has an undefined result when its argument is an empty stack. Rather than simply leave this case undefined, VISAGE allows the specifier to indicate explicitly that it results in an error. This has the benefit that users are left in no doubt about the result of applying an operation, something particularly useful for the anticipated users of VISAGE who are inexperienced at specifying ADTs.

To allow the explicit specification of error situations, VISAGE has a special nullary term called `error!` that can appear as the sole term on the right-hand side of an equation e.g. `top(new.stack) = error!` for the stack example above. The `error!` value belongs in every sort within VISAGE. Unfortunately, this mechanism is too simplistic to cope with the subtleties of handling general error conditions in specifications as discussed in Goguen et al. (1978). However, by restricting occurrences of the `error!` term to the right-hand side of equations, many of these deficiencies can be reduced.

If an operation is partial it seems appropriate that this information be made known in the Signature View since it is this view that is responsible for describing the operations of an ADT at the most abstract level. At the moment, the arc leaving an operation icon is directed towards the icon representing the sort of its range. This connection is shown as a thick, solid directed line. Partial operations could be naturally represented by a different line style, for example, a dashed rather than solid line. The different output links are illustrated in figure 5.10, where the data icon represents the parameterised sort of value stored by the stack.

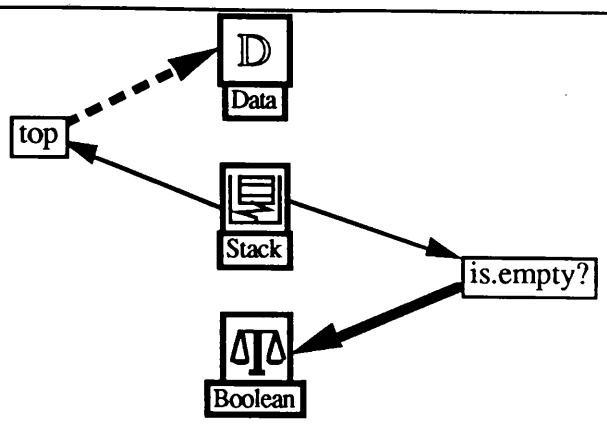


Figure 5.10: Representations of Partial and Total Signatures

In figure 5.10, the `is.empty?` operation has its result link shown as a solid line indicating that it is a total operation. In contrast, the `top` partial operation has its result link shown as a dashed line.

5.9 Equation View

An early idea for showing the behaviour of operations was to modify the signature diagram so that it could represent the equations of the specification as well as describe the signatures of the operations involved. To illustrate this, consider the equation `top(push(d,s)) = d` for the stack ADT example. The signature diagram of figure 5.7 has been partially broken up to leave the templates of the `push` and `top` operations shown in figure 5.11.

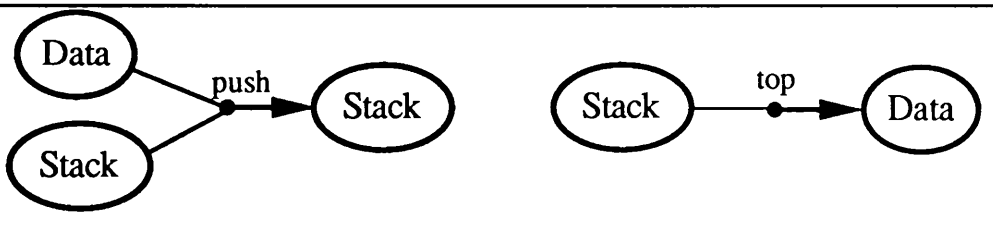


Figure 5.11: Operation Templates for `push` and `top`

Instead of representing carrier sets as in a signature diagram, the ovals in figure 5.11 are unnamed variables that can be assigned values of the named sort. These templates can now be joined together into a data flow circuit so long as input and output termini of the same sort are connected together. Since the output of `push` has the same type as the input of `top`, these two templates can be joined to form figure 5.12.

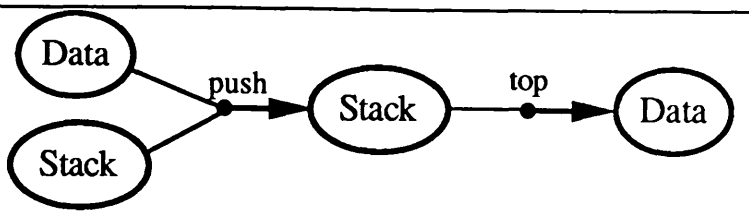


Figure 5.12: Joining Equation Templates

This dataflow circuit could represent the term `top(push(d,s))` where `d` and `s` are variables of sort data and stack respectively. To complete the representation of the equation requires showing that the output of the overall circuit is the original data input. This could be achieved as shown in figure 5.13(a). This diagram can be further simplified by removing the intermediate stack value representing the result of the `push` operation. The dataflow nature of the diagram could also be emphasized by placing arrowheads on the input links of an operation. These changes result in the first attempt of representing an equation using a modified signature diagram, and is shown in figure 5.13(b).

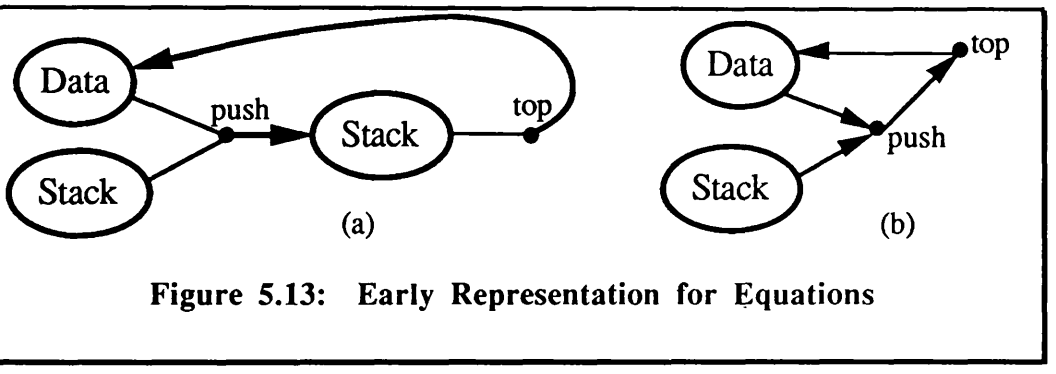


Figure 5.13: Early Representation for Equations

The representation shown in figure 5.13(b) clearly shows that the value held in the stack variable is immaterial and does not affect the action of `top` on a `push` term. However, although this ad hoc representation seems to work for the above examples, it only works when the right-hand term is a variable that appears on the left-hand side of the equation. As an example of the difficulty, consider the equation below:

```
null (push (d, s)) = push (d, s)
```

that states that the operation `null` has no effect when applied to a `push` term. To handle this example, the graphical notation has to be extended to provide a way of referencing the implicitly defined value `push(d,s)`. This was done by the introduction of the concept of an *anonymous variable* that stores the intermediate result. Although these variables are anonymous they are uniquely identified by their position within the view. The graphical depiction of the above equation is shown in figure 5.14.

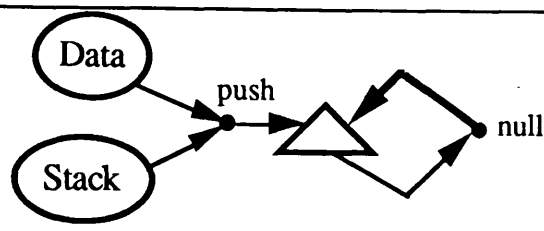


Figure 5.14: The Use of an Anonymous Variable

In this figure, the triangle is the graphical representation of the anonymous variable and is considered as holding a single, constant value: in this case, generated by the `push` term since the diagram should be "parsed" from left to right. All terms "flowing" into the variable have the same value. This value is also used in the above figure by the `null` operation whose result link feeds back to the anonymous variable. Given the constraint that the variable can have only one value†, it can be inferred that the `null` operation is simply the identity function (its output being the same as its input).

This general mechanism of having an anonymous variable constrained to hold a single value can also be used as a way of representing the fact that the two sides of an equation must return the same value. There is no need for the value stored in the anonymous variable to be used as an argument. Consider for example, the equation that states that a newly-created stack is empty: `size?(new.stack) = zero` and is represented as shown in figure 5.15:

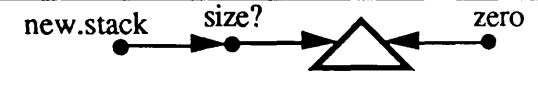


Figure 5.15: Representing Equality in Equations

Although this extended graphical notation, based on the original signature diagram, is now capable of representing arbitrary equations, it still suffers from a number of deficiencies. For example, the use of ovals to represent variables of the corresponding sort can cause ambiguity when there is more than one variable of the same sort in an equation. To overcome this restriction, variables are now represented as icons with an image representing their sort and their name appearing in the label. This form allows the user to identify the variable each icon denotes as well as the sort of its value. To promote consistency between VISAGE's different graphical views, the dot representation of operation instances is replaced by the iconic form used in the Signature View. In addition,

---

† Perhaps the name 'anonymous constant' would be more appropriate. However, since a value is assigned the concept of a variable is closer than that of a constant, hence the adopted terminology.

since the output of operations is obvious from the arrows on the links, the output links do not need to be distinguished from the input, using thicker line styles, as in the Signature View. However, in keeping with the Signature View, operations with more than one argument have their argument links numbered with their position in the formal parameter list. Given that the orientation of an equation is important when considering it as a rewrite rule (i.e. which term is on the left-hand side of the equation?), the graphical representation of an equation should also convey this information. This is handled simply by drawing the left-hand term to the left of the right-hand one! After making these changes, the graphical representation of an equation such as

```
size? (push (d, s)) = succ (size? (s))
```

is shown in figure 5.16.

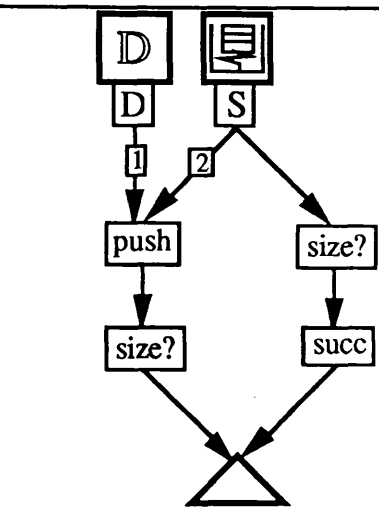


Figure 5.16: Graph Representation of an Equation

Although the representation shown in figure 5.16 worked for arbitrary equations, automatic layout of the graph was complicated by the positioning of variables. This representation tried to enforce the idea of a variable being a unique storage location: if that variable appeared in several places in the equation then links would connect its icon to all its users. For example, the variable `s` in the figure has links connecting it to the two terms that use it. The icon for variable `s` must be positioned so that it is equally accessible to the `push` and `size?` operation icons. Although this is relatively simple in the above example, the problem rapidly becomes intractable with more variables and instances. To overcome this "spaghetti ball" problem, the representation was modified to create an icon for each variable instance. This simplifies the layout algorithm as it now deals with tree structures rather than graphs: each icon is juxtaposed with its user. By highlighting all icons for instances of the same variable whenever one if instance is selected, the user is continually reminded that the instances are occurrences of the same variable. Figure 5.17 below shows the final dendritic representation of the equation.

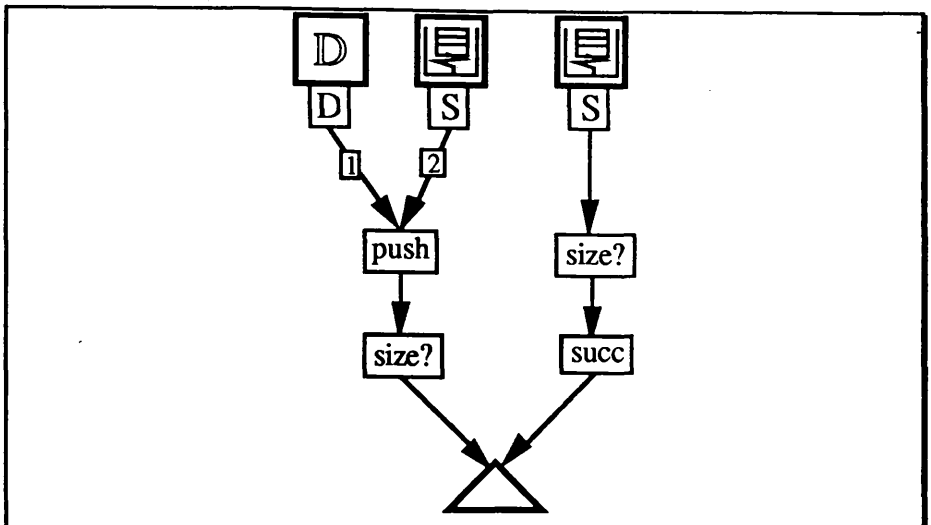


Figure 5.17: Final Tree Representation of an Equation

Early concern that having an icon for each variable instance would confuse the user were unfounded judging by users' reaction during the evaluation (described in chapter seven). Additional benefits of having a tree representation are discussed in a later section on the visual programming of equations.

5.9.1 The Layout Algorithm

The Equation View represents an equation graphically using a tree format, an example of which is shown in figure 5.17. The heart of the layout algorithm is the mechanism for generating the layout of a term. The operators that make up the term are laid out vertically into levels. The root of the term is placed at level one; the term's arguments are assigned to subsequent levels depending upon the distance of the node from the root of the term tree which corresponds to the degree of textual nesting involved. For example, table 5.3 shows the level assignments for the term: `top(push(grow(new),pop(s)))`.

Level	Operators
1	top
2	push
3	grow, pop
4	new, s

Table 5.3: Levels of Operators in the Equation View

These levels can be thought of as horizontal rows with the option of level one being either the top or bottom row in the diagram, as desired, thereby giving the user the option of drawing the term trees according to Computer Science convention or as nature intended!

The screen distance between the rows is automatically adjusted to ensure that links between icons adjoining levels can be easily discerned.

Once the icons for the operators have been assigned their Y-coordinate, the algorithm then deals with their arrangement along the X-axis. To explain the algorithm, consider the case of the general term `t(s1,...,sn)`. The layout procedure is recursively applied to the sub-terms s_1 to s_n in order. The sub-terms are given a starting X position that ensures that adjacent sub-terms do not overlap. Each invocation of the layout procedure returns the X position (known as the *placement)* assigned to the root icon of that sub-term. If the sub-term is a leaf node then the placement is simply the starting position. Once all the sub-terms have been placed, the root icon for the operator `t` is placed at the average placement of its sub-terms $s_1,...,s_n$ i.e. midway between the placements of the first and last sub-terms. A schematic representation of the layout procedure is shown in figure 5.18.

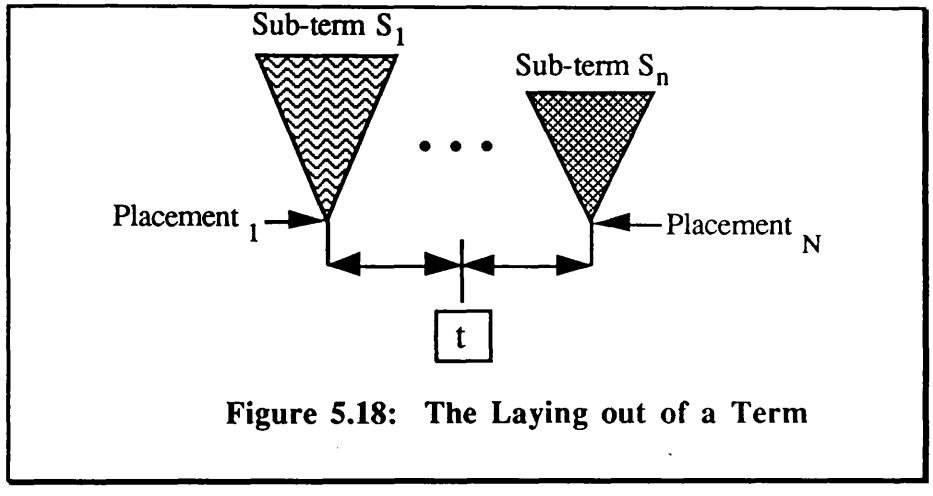


Figure 5.18: The Laying out of a Term

This algorithm is illustrated using the left-hand term of the equation shown in figure 5.17 as an example: `size?(push(d,s))`. Given the recursive nature of the algorithm, the first icon to be placed is for variable `d` which is assigned position A. Variable `s` is then placed at position B. Their root operator `push` is placed midway between them at position C = (A + B)/2. Since `push` is the only sub-term of `size?`, the `size?` icon is also placed at position C.

In laying out an equation, the algorithm works by regarding it as a term with the root operator being the equality constraint between the two sides. Once the terms have been laid out, links between the operator icons showing the nesting relationships can then be drawn.

5.10 Operation Dependency View

The graphical views described so far allow the user to select individual ADTs to visualise, and then select signatures and equations within an ADT specification. Although these views allow the user to examine all the component parts of a specification they do not let

the user *explore* the relationships that exist between the operations of an ADT. Unlike procedural descriptions, an algebraic specification of an individual operation's behaviour is not localised, but instead is spread over a number of equations. The operations that appear in these equations are obviously related, and a tool to help investigate these relationships is obviously desirable as a way of understanding the behaviour of the operation. Unfortunately, such a facility is not present in the usually primitive, textual specification environments available today: the Operation Dependency View is a first attempt at filling this deficiency within an interactive graphical framework.

Before the Operation Dependency View can be implemented it is necessary to have a clearer understanding of what it is trying to visualise. To simplify the description, consider the example equation below:

```
is.empty? (push (d, s)) = false
```

This equation states that pushing a new element onto any stack results in a non-empty stack. This equation involves three operations that are related in some way and can be grouped together into a set: `is.empty?`, `push` and `false`. This distillation process, when applied to all the equations in a specification, generates a set of sets of related operators. To illustrate this, consider the equations below for the stack ADT and the sets they produce:

```
1.  is.empty?(new.stack) = true    { is.empty?, new.stack, true  }
2.  is.empty?(push(d,s)) = false   { is.empty?, new.stack, false }
3.  pop(push(d,s)) = s             { pop, push }
4.  pop(new.stack) = new.stack     { pop, new.stack }
5.  top(push(d,s)) = d             { top, push }
6.  top(new.stack) = error!        { top, new.stack, error! }
```

Note that a single operator can appear in more than one set, and although it is possible for two sets to be identical, it is a rare occurrence. However, if this situation does arise, a menu of the resultant equations will be displayed, from which the user may select the desired equation to view. The Operation Dependency View therefore allows the user to select an equation by selecting its corresponding set of operators: this mechanism is discussed fully in the next section.

5.10.1 Interactive Exploration

The basic action in exploring the relationships is the selection of an operation's icon (implemented as clicking on the icon, say) which adds the corresponding operator to a special selection set. Whenever a new operator is added to the set, the view disables all equations (by displaying them in a dimmed style) that do not have instances of *all* operators in the selection set. By repeatedly selecting operation icons the number of avail-

able equations is reduced until only one remains. This process of refining an equation selection can be reversed by removing an operator from the selection set (by simply deselecting it).

To help illustrate this mechanism, imagine a user wishes to understand how the `push` operation behaves in the stack example. Assuming that the selection set (call it S) is initially empty, the user selects the `push` operation giving S = {`push`}. The effect of this selection on the Dependency View is shown in figure 5.21. VISAGE also disables all equations that do not involve all the contents of S, leaving the following equations:

```
2.  is.empty?(push(d,s)) = false
3.  pop(push(d,s)) = s
5.  top(push(d,s)) = d
```

Note that since the operators `new.stack`, `error!` and `true` do not appear in this reduced set of equations since adding any of them to S would mean that no equation would be selected. To avoid this situation the view prevents the user from selecting operators that do not appear in any of the currently selected equations.

Having made this initial selection the user can refine it by selecting another operator. Say, for example, the user wants to know how `push` behaves in conjunction with the `pop` operation: by selecting the `pop` operation, S = {`pop, push`} and the only equation that has instances of both these operators is `pop(push(d,s)) = s`. Note that the order of adding operators to the selection set is not important. As the user selects operation icons, VISAGE ensures that the other displays (textual and graphical) update their display in an appropriate way to reflect the contents of the selection set. In particular, once the user has selected a unique equation VISAGE will have the Equation View display it. Correspondingly, if the user selects operations or equations in other views then the Operation Dependency View will automatically highlight the relationships involved.

Once the user has seen the relationship between `push` and `pop`, removing operators from the selection set (implemented as a second click on the appropriate icon) lets the user backtrack and explore other relationships. For example, deselecting `pop` and then selecting `is.empty?` will cause the equation `is.empty?(push(d,s)) = false` to be selected and displayed as it is the only equation involving both `push` and `is.empty?`. Note that the user could have instead deselected the `push` operator thereby starting an exploration of those equations involving `pop` operator.

5.10.2 The Layout Algorithm

The main design problem in the Operation Dependency View amounts to finding the best way of presenting the set of sets of related operators as clearly as possible. The obvious approach is to use Venn diagrams whereby each set is represented by an enclosed spatial

area, with intersection of areas denoting those elements that belong in more than one set. Unfortunately, the large number of intersections together with the difficulty of automatically drawing area boundaries made this approach impracticable.

The next alternative simplifies the representation to a node-and-arc display with different styles (or colours, preferably) signifying which nodes belong to the same set. The result of applying this to the sets produced by the stack equations is shown in figure 5.19.

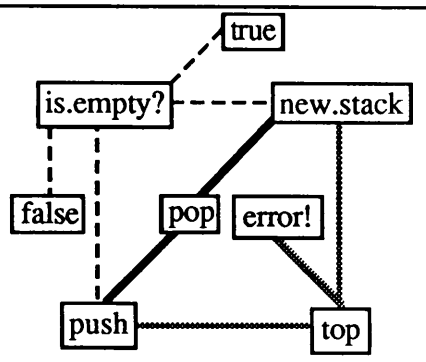


Figure 5.19: Operation Relationships – a First Attempt

The first attempt at visualising these graphs took the simplifying assumption that all the operations in a specification are equally important and so they should be displayed in an egalitarian manner. The approach taken was to place the operation icons equidistantly around the circumference of a suitably sized circle. Arcs would then be drawn between the icons of operators in the same set. Although simple and quick to display, this approach suffered from several disadvantages. Firstly, as the number of operations increases, the circle expands in proportion until the point is reached when it cannot be displayed in its entirety within the view, requiring that the user scroll the view to reveal the hidden parts. Using alternative display geometries, such as spirals, generally resulted in pictures with inter-connection arcs resembling a spider's web after a storm: these alternatives were abandoned. A further problem was that as the specification grows it becomes increasingly difficult to generate distinct line styles, especially as this version of Smalltalk did not support a colour display.

The final alternative overcomes these problems by dispensing with the display of the arcs altogether. This simplification arose after it was realised that a static representation is unnecessary given that the view will always be used as a way of *interactively* exploring the relationships between operations. If the view can change dynamically to reflect the selections made by the user, then the complexities of producing a single static representation can be forgotten. All that is necessary to represent the state of the search space at any one time is a mechanism for partitioning the operators into three groups:

1. selected by the user,
2. disabled given the current selection set, and
3. neither selected nor disabled.

Many visually-based solutions exist to this problem. For example, the view could be split into three bounded areas with icons migrating between the areas as appropriate for a given user selection. An alternative approach is taken in VISAGE because, as will be discussed shortly, it could be combined with an additional dimension of information to produce a richer representation.

The three different partitions an icon can belong in are encoded using graphical highlighting techniques. If an operator has been selected then its icon is shown in reverse video; a disabled operator has its icon shown dimmed; otherwise the operator's icons is displayed in the normal fashion. These different visual states are shown in figure 5.20.

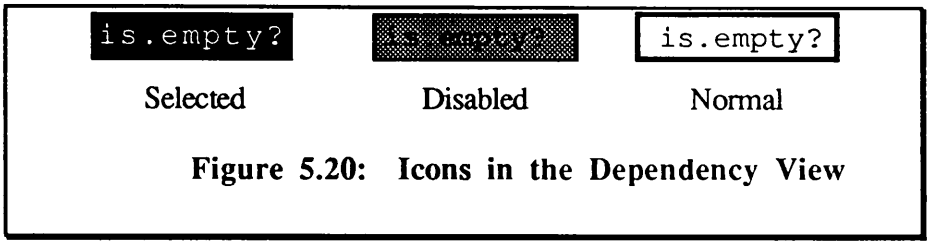


Figure 5.20: Icons in the Dependency View

In displaying the icons, the view can present additional information if the icons are not simply presented as an amorphous collection. By adopting spatial encoding techniques, the view can simultaneously show the current state of the search space and a useful organisation of the operators involved.

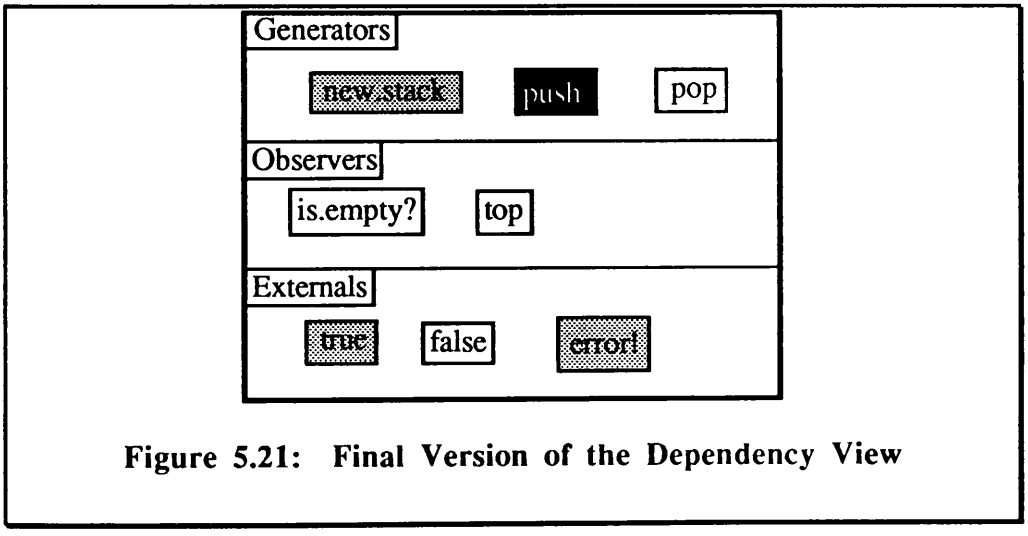


Figure 5.21: Final Version of the Dependency View

The obvious contender is to partition the operators into observers, generators† and those defined outwith the currently selected ADT. This can be done by a simple analysis of the

† These terms are described in chapter two.

current ADT's signature. When combined with the selection display mechanism, the final version of the Dependency View appears as shown in figure 5.21. The example shown in this figure is the state of the display after the user has selected the `push` icon, thereby disabling the icons for the operators `new.stack`, `true` and `error!`. The selection can be refined by selecting one of `pop`, `is.empty?`, `top` or `false`.

5.11 Summary

This chapter has introduced and described the various textual and graphical views provided within VISAGE to represent the various facets of an ADT's structure and behaviour. Each view's description included a detailed account of its display algorithm and development. These views can completely describe ADT specifications written using the VISAGE textual specification language.

The following chapter concludes the description of VISAGE by discussing the extension of the above visualisation machinery to facilitate the graphical construction and editing of an ADT specification. The chapter also describes the PLAYPEN, an animated, graphical term rewriting environment, integrated with the rest of VISAGE, that allows users to create example data and have them transformed by application of the equations that define the behaviour of an ADT.

Chapter Six
Visual Programming and Experimentation

6.1 Visual Programming of Specifications

The purpose of visualisation is to map a complex, abstract object such as an ADT, onto one or more, concrete (graphical) representations in order to make the behaviour and structure of that object more understandable. A logical question is whether these concrete representations would be a good way of creating the abstract objects in the first place i.e. is a good program visualisation representation necessarily a good visual programming one? The VISAGE system was originally developed as an experiment in visualising algebraic specifications. However, once this facility had been implemented, it became clear that graphical representations that were successful for output should also be successful for the *input* of algebraic specifications. This section describes the results of extending the VISAGE prototype to support the creation and subsequent modification of algebraic specifications of ADTs using the graphical representations already developed, i.e. extend VISAGE to become a visual programming language.

Although VISAGE is being extended to allow the visual programming of specifications it should be remembered that a textual editing facility has been provided since the earliest version of the system. The graphical editing described in this section aims to complement this facility and be tightly integrated with it. As discussed in chapter three, synchronising textual and graphical editing offers the chance for users to acquire an understanding of one representation through the manipulation of a different one.

The views developed for VISAGE to visualise formally-specified ADTs each show a different aspect of an ADT. Given this partitioning of facets to different views, initially it seemed likely that each view will have to be given an editing ability so as to create and modify the facet they display. This section will develop a graphical editing facility based on this belief and will identify any views that are superfluous for editing, as well as those that are deemed necessary but which were not needed for visualisation.

One of the major differences between traditional programming support and the more recent programming support environments is removal of the modes that demarcate the editing, compiling and execution phases of development: the user appears to be continually in an editor of some description. A design goal for VISAGE is to provide just such a seamless environment for graphically building and browsing ADT specifications. The intention is

that by removing the distinction between visualising, programming and then executing specifications, the novice user does not experience the sort of problems commonly encountered with the traditional, batch-oriented style of interaction. A typical problem is forgetting to compile a newly edited program only to be confused by the discrepancy between expected and actual run-time behaviour of the program. In VISAGE, the user should be able to browse, program and execute specifications in any order they desire, without the expense of switching modes, e.g. by switching to another tool. This facility, when combined with support for incomplete specification, will hopefully promote an incremental, "try-and-see" style of specification that ensures results are generated quickly, providing strong positive feedback to a novice user.

6.1.1 The Editing Mechanism

The visualisation mechanism outlined in chapter five may be regarded as a mapping from the internal abstract syntax tree (AST) formed by parsing a specification to a set of external views that illustrate various properties of this structure. The visual programming facility will perform the inverse action, converting the creation and manipulation of graphical objects into changes to the AST. This symmetry is shown in figure 6.1.

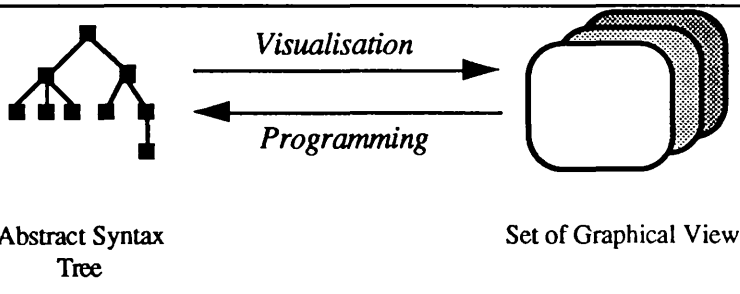


Figure 6.1: Program Visualisation and Visual Programming

This bi-directional mapping is possible because there is a one-to-one correspondence between graphical entities and segments of the AST. The creation of a new graphical object results in the creation of new syntactic object. Links between graphical entities correspond to branches in the abstract syntax tree and cause the newly connected object to be grafted onto the main syntax tree. Conversely, deletion of graphical objects corresponds to pruning of the tree.

6.1.2 Enforcing Correctness

In traditional programming (and specification) environments the user typically creates a program† using a free-form text editor and when finished, passes the result to an appropriate processor for checking and perhaps compiling into some other, more efficient (by some criterion) representation. Unfortunately, it is rare for the user to have produced a totally correct program at the first attempt with the result that several edit–compile cycles are required before an acceptable program is produced. Even when a program compiles successfully there may still be problems when it comes to integrating the program with the rest of the environment. This iterative process is partly due to the separation of the editor, compiler and environment: the compiler cannot help the editor in constraining the user to enter only legal programs; and the environment cannot help in ensuring global consistency among its component parts. Considerable improvement in the development of programs would result from the tight integration of the editor, compiler and environment. This has been demonstrated for programming by the Smalltalk system (among many others): the desire is to achieve it for algebraic specification within the graphical world of VISAGE.

Rather than let the user create or do something and then complain that it is invalid, it is much more sensible and efficient to prevent the user from performing an illegal action in the first place. This incremental checking simplifies specification development since it deals with the user's current task rather than having them try to understand an error from a different context as is typical with batch processing. This can be achieved using simple and unobtrusive techniques within a graphical system. For example, highly interactive interfaces, and particularly those of a graphical nature, often provide *semantic feedback* on the user's actions, thereby giving a direct indication of what is legal in a particular circumstance. Such feedback allows the user to develop quickly a conceptual model of what can be done to different objects. Semantic feedback is heavily used in VISAGE appearing in several different guises.

Firstly, if all actions are activated by means of a menu command then illegal commands, for a given situation, can be avoided by simply omitting them from the menu. By presenting only legal command options, these context-sensitive menus can serve as a tutorial guide on the system's facilities, helping the user select appropriate commands.

In the node–and–arc world of VISAGE, a common editing action is to connect two nodes together with a directed arc. In keeping with the Direct Manipulation paradigm adopted for VISAGE, this editing action is performed by drawing a line from one node to the other.

---

† The terms 'program' and 'compiling' should be taken to cover formal specifications and their mechanical processing, e.g. semantic checking or execution.

Semantic feedback can help in two ways with this operation. Firstly, while the user is drawing the line it is reassuring to show him that the system is correctly interpreting his actions. This reassurance is provided by the use of a "rubber-banded line" anchored at the starting node and tracking the movement of the cursor. The user selects the destination node by clicking on it. However, what happens if this link is nonsensical or illegal? The system would have to issue an error message requesting the user not to be so silly: not very friendly. A better approach would use semantic feedback to inform the user what nodes would be acceptable as an end-point for this link. This could be done in several ways, e.g. the display could gray-out and disable all unacceptable nodes; alternatively, the system could flash the icon if the cursor is over a legal end-point for this link.

6.1.3 Generic Editing Protocols

In designing VISAGE's editing graphical facility it was essential to adopt a consistent and intuitively simple protocol so that users will find it convenient to use and easy to remember during a session and hopefully between sessions.

Graphical structures representing specification components can always be created in VISAGE by simply creating new nodes and adding links between them. Unfortunately, even such a simple editing technique as this causes considerable difficulty when it comes to designing the optimal user interface for the task. Perhaps more sophisticated editing techniques need to be developed within the node–and–arc framework to let users quickly build structures without having to use complex sequences of low-level operations to achieve their goal. The only approach available given the current state of interface design theory is to experiment with different approaches, evaluating (either formally or informally) their relative effectiveness. Fortunately, Smalltalk's suitability as an environment for rapid prototyping allows a variety of approaches to be tested in a comparatively short time.

All editable graphical views in VISAGE have commands for adding and deleting icons and links. The following sections describe the basic protocol developed to handle this graphical editing. Although the protocol is similar in all the views, the semantics of the editing action is view-dependent. The more sophisticated editing facilities available for manipulating equations and terms will be discussed separately.

Adding New Icons

In each editable view, the menu that appears when the middle mouse button is pressed with the cursor over the view's background includes the command to add a new icon to that view. The command asks the user to provide an identifier for the object to be created. The system ensures that where necessary the identifier is unique, and if it is not, the user is informed about the conflict and then given the chance to supply an alternative identifier or

to cancel the operation. It is essential that the user always has the chance to cancel, without penalty, an operation that has encountered problems.

Identifiers in VISAGE must start with a letter but can be any non-zero length. The identifier can use letters, digits and a variety of punctuation and other printing characters. This relaxed naming convention avoids the problems experienced by novice users when using systems that impose unnatural restrictions e.g. the infamous *at most six capital letters or digits, and starting with a letter from I to N* for integer variables in FORTRAN.

Although the user can supply a name using either upper or lower-case letters, VISAGE will automatically convert all letters to lower-case. This is done for two reasons. Firstly, novice users often type identifiers with mixed case e.g. name and Name. If these refer to different objects then the user will probably be confused by the subsequent behaviour of the system. Secondly, these identifiers are used as keys in VISAGE's internal dictionaries for accessing objects. By converting everything to lower case, look-up problems due to case mis-matches are avoided.

Once a satisfactory name has been supplied, an iconic image is automatically created for this object and "attached" to the cursor ready for the user to place on the view. The image used depends on the kind of object being added. When the cursor is at the desired location for the icon, the user clicks a mouse button to "drop" the icon there. The cursor is constrained to stay within the bounds of the view. This completes the creation of a new icon.

The reason for letting the user position the icon is due to the fact that automatic positioning, as used in an early version of VISAGE, was found to confuse and frustrate users. Since users prefer having control over an icon's position, it seemed sensible to let them supply its initial position, with automatic re-positioning being an option that must be explicitly requested.

Adding New Links between Icons

A directed link is used to represent a relationship between the two entities it connects. VISAGE allows the user to create new links in the Hierarchy and Signature Views using the ADD LINK command†. This command is on the menu available when the user presses the menu mouse button with the cursor over an icon that would be a legal starting point for a link. Once the command has been issued, the user moves the cursor until it is over the end-point for the link. A rubber-banded line, anchored at the link's starting point tracks the cursor location until the user clicks a mouse button to complete the link. If the cursor was over a valid end-point icon then the link is established otherwise the effect is to cancel

† Links in the Equation View are created automatically as a side-effect of other editing actions.

the ADD LINK operation. To ensure that only valid links can be created, the system uses graphical semantic feedback to guide the user: all the icons that would be illegal end-points given the current starting point for this link are disabled (i.e. cannot be selected) for the duration of the link creation task. This mechanism is discussed fully in the section above on enforcing correctness. As with adding new icons, the effect of adding a link is view-dependent.

Deleting Icons and Links

To delete either an icon or a link the user simply points at the object to be deleted and issues the DELETE command from the menu. Pointing at an icon is simply a matter of positioning the cursor within the icon's screen image. However, links are more interesting. The obvious approach is to let the user select a link by pointing at any point on its path. Unfortunately this simple approach is too inefficient. Since links are drawn as cubic splines, the computation required to determine whether the cursor is close enough to a point on the path is excessive. This problem can be avoided by insisting that the user select a path by pointing at a unique point on its path.

Since the link's end-points are often shared with other links they are not suitable as unique reference points. The approach used in VISAGE is to make a small area around the link's arrowhead the sensitive zone for selection by the cursor. This zone is shown as a gray rectangle in the example link above. The area of this zone can be varied depending upon the screen resolution and users' manual dexterity. This approach is simple, easy to remember and efficient for selection.

After the delete command has been issued for a particular object, the system will perform the appropriate action including requesting any user confirmation, and repairing the data-base where necessary.

6.1.4 Editing in the Hierarchy View

Creating New ADTs

Since the Hierarchy View is the only one dealing with an ADT as a single entity, it seems the natural place to put the facility to create new ADTs. The act of creating a new ADT would be viewed as adding a new icon to the Hierarchy View. Before a new icon is created, the user must supply an identifier for it. This name is used to declare the new ADT and must therefore be unique within the current environment: a constraint that it directly imposed by the system thereby ensuring that name clashes will not arise.

Once the icon has been named, the user is invited to position it within the view. Thereafter it can selected for viewing just like any other ADT. A newly created ADT will be

displayed in the textual Specification View as an empty, named template; for example, the specification of an ADT called new will look like this:

```
Datatype:    new;

Equations:
EndSpec
```

Subsequent graphical editing actions (in whatever view) will be reflected automatically in the textual view, and vice versa.

Creating New Usage Dependencies

The other major editing action to perform in the Hierarchy View is the creation of links between icons to represent relationships between the corresponding ADTs. A directed link is added in the standard way with the direction of the link denoting which ADT is using which (as described in the section on the Hierarchy View's representation). However, to ensure that the new link will not introduce an inconsistency into the ADT database, VISAGE first determines which ADT icons would make valid end-points for a link starting at the current ADT icon. An icon is considered invalid if a link to it would introduce a circularity in the usage dependencies. All invalid icons are temporarily disabled (their images are dimmed) during the life-time of the ADD LINK operation, thereby making it impossible for the user to introduce a circularity.

Deleting ADTs and Usage Dependencies

Just as creating ADTs and usage dependencies is done by creating new icons and links in the Hierarchy View, so deleting ADTs and breaking dependencies involves deleting icons and links. However, care must be taken to ensure that such deletions are actually meant (done by confirming the action with the user) and that they do not leave the ADT database in an inconsistent state.

Firstly consider the effect of deleting an ADT icon. If that ADT is not connected to any other (i.e. there are no usage dependencies involved), or is not being used by any other ADT, then it can be deleted with impunity. However, if this ADT is being used by others then some repair of the database will be necessary before it is deleted. To illustrate the repair mechanism, assume that the ADT to be deleted, call it A, is being used by a set S of other ADTs. For every ADT in S, all the operations that mention A in their signatures must be deleted; this in turn demands that all equations that use instances of a deleted operation must in turn be deleted in this ADT *and* in all ADTs that use it. Given the global effect of this action the user is prompted for confirmation before the deletion is performed. Once all these deletions have been made, the ADT A can be removed from the database. A

similar repair mechanism is required when deleting a usage dependency link, but instead of a set of ADTs requiring repair, only the ADT A is involved.

In the original VISAGE prototype, if ADT A used B and C, and B used C, then the link from A to C was omitted as it was implied by transitivity. This approach was used as it minimised the number of links that appeared in the Hierarchy View. However, when VISAGE was extended to allow visual programming, this representation made it impossible to delete the dependency of A on C as it had no explicit representation. The original representation used by the Hierarchy View was ambiguous in that it could not distinguish between $A \rightarrow B \rightarrow C$ and $C \leftarrow A \rightarrow B \rightarrow C$ (where the arrow represents the usage dependency). To overcome this problem, the Hierarchy View was modified to show all links.

6.1.5 Editing in the Signature View

Just as the Hierarchy View is the natural place to create and manipulate ADTs, so the Signature View is the natural place to create and manipulate the operations belonging to a particular ADT. When an ADT is selected, the Signature View displays the operations associated with it, together with their signatures i.e. their relationships with the various sorts available within this ADT.

Following the editing protocol discussed above, declaring new operations is simply a matter of creating new operation icons and drawing links between them and the desired sort icons to define the operation's arity. However, just as the Hierarchy View places a specialised interpretation on this general protocol, so does the Signature View.

Creating New Operations

When creating a new operation the user is asked to supply an identifier for the operation and then to position it within the view. The only constraint on the identifier is that it must be unique within this ADT: VISAGE allows the same identifier to be used for the names of operations in other specifications without conflict. Once the operation icon has been named and positioned, a textual representation is shown in the Specification View. If, for example, the newly-created operation is called `oper`, then the signature would appear as: `oper: → unknown!.` This default signature states that `oper` is (currently) a nullary operation with an unknown range. The symbol `unknown!` is a special, pre-defined sort-name that is used as a place-holder to ensure that the signature is always of the correct form: it will be replaced later by a real sort-name when the user edits the operation's signature. Once an operation's signature template has been created, the user can edit it into the desired arity.

Editing an Operation's Signature

Whenever an ADT has been selected in the Hierarchy View, the Signature View shows icons representing the (carrier set of the) sorts that appear in that ADT's specification thereby ensuring that the user has access to all the sorts needed for defining a signature. The user defines an operation's signature by adding (and optionally removing) links in the Signature View.

The ADD LINK command is used to add both argument and result links for operations: if the link is drawn going to the operation icon then it is assumed to be an argument; a link leaving the operation icon is deemed to be the result link. As each link is added, the Specification View updates the textual representation.

When creating argument links for the operation (i.e. defining its domain) the View automatically numbers the links (when there is more than one) to identify their position in the argument list. Similarly, if an argument link numbered m is deleted from a set of n links ($m \leq n$), the links numbered m+1 to n have their numbers decremented so as to fill the gap.

The Signature View differs from the traditional signature diagram in giving each operation its own argument and result links even if the signature is the same as another operation's. For example, figure 6.2 shows how the two signatures:

```
add: integer × integer → integer
subtract:  integer × integer → integer
```

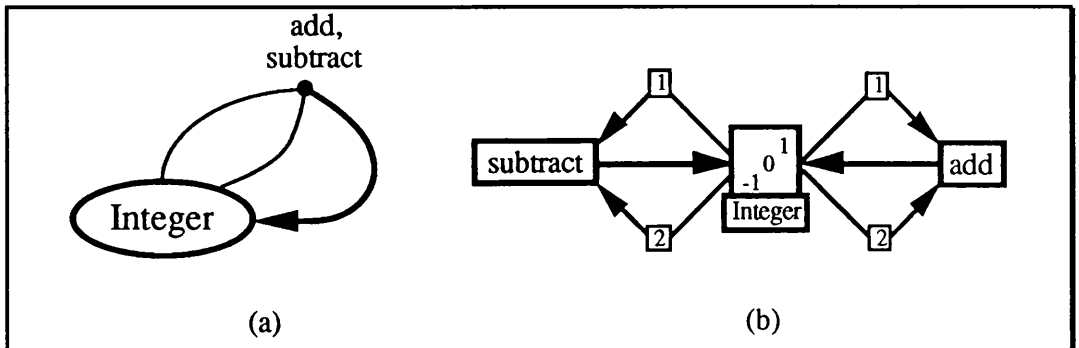


Figure 6.2: Combined and Individual Graphical Signatures

would be shown in a signature diagram (figure (a)) and the Signature View (figure (b)). The advantage of the Signature View's representation is that editing is now much simpler, e.g. it is simpler to add and remove links to an individual operation icon without involving display updates that are both tricky to compute and hard for the user to follow.

If instances of the operation being edited appear in any equations in this ADT or any that use it, then the system must modify these equations so maintain type consistency. For example, given the specification fragment shown below on the left:

```
....
build: data → struct
....
Equations
size? (build (d)) = big
....
```

```
....
build: data × struct → struct
....
Equations
size? (build (d, unknown!)) = big
....
```

if a new argument link of sort `struct` is added to `build`'s signature, this will change the fragment to become that shown on the right above. An `unknown!` place-holder term is automatically inserted into the equation to make the actual argument list consistent with the formal parameter list given by the operation's signature. The constant `unknown!` is polymorphic and can appear wherever a term was expected. Similarly, if an argument link is deleted in the Signature View, all equations that have instances of the operation involved have the sub-term corresponding to this argument deleted (after user confirmation). A more drastic repair is needed if it is the result link of an operation that is being deleted. In this situation, all equations that use this operation have their instances of this operation replaced by the special `unknown!` place-holder term. As an example, if the result link for the `build` operation introduced above is deleted, the specification fragment on the left below would change into that on the right:

```
....
build: data × struct → struct
....
Equations
size? (build (d, s)) = big
....
```

```
....
build: data × struct → unknown!
....
Equations
size? (unknown!) = big
....
```

The *VISAGE Tutorial*, included as appendix D, gives a step-by-step guide to the definition of example operations for the stack ADT using the graphical editing commands available within the Signature View.

6.1.6 Editing in the Equation View

The Equation View differs from the Hierarchy and Signature views in having a richer set of editing facilities for manipulating terms and equations. To explain these facilities it is useful to introduce a metaphor. This metaphor also proved very helpful in the design of these editing facilities.

The *Plug and Socket* Metaphor

Central to editing in the Equation View (and its relatives like the PLAYPEN to be described shortly) is the *Plug and Socket* metaphor. Every icon in the view (except the special icon representing the equality constraint of the equation) is regarded as a *socket* into which can be inserted a *plug* of the corresponding "shape". These sockets are "magical" in that a plug can be inserted into a socket even if there is already a plug there: the new one simply replaces the old. A plug is simply an unconnected iconic structure representing a sub-term that is being moved. The metaphor uses the physical property of shape to denote the abstract property of an object's type. The physical compatibility between the shape of a plug and a socket that will accept it, mirrors type compatibility in the computational domain. For example, a socket of "shape" stack will not accept a plug of "shape" Boolean. An extension to the implementation of this metaphor allowing geometric shape to represent an object's metaphorical shape (i.e. its type) is discussed in chapter eight on future work.

Creating a New Equation

Editing in the Equation View involves building tree structures representing terms by plugging together simpler structures based on templates created using menu commands. To illustrate these facilities consider the creation of the equation†:

```
is.empty? (push (d, s)) = false
```
— Equation 6.1

A new equation is created using the ADD NEW EQUATION command which has the textual representation `unknown! = unknown!` as shown in the Specification View. This embryonic equation appears in the Following graphical template:

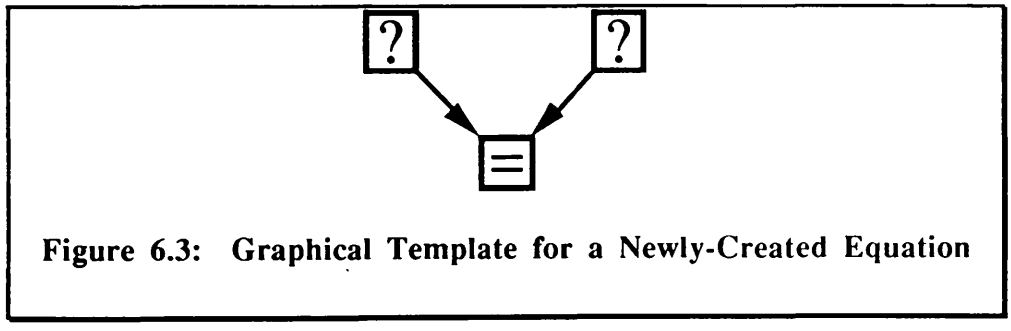


Figure 6.3: Graphical Template for a Newly-Created Equation

The icon at the root of this tree (shown with the equals sign) is an anonymous variable that constrains its left and right-hand sub-trees to produce the same value. The two icons with the question-mark images are the sockets for the left and right-hand terms of the equation. These icons represent the special term `unknown!` and their image tries to convey the fact

---

† A step-by-step example of creating complete equations graphically is given in the *Visage Tutorial* reproduced in Appendix D.

that they will accept a plug of any shape i.e. structures representing terms of any type can be plugged into these sockets.

Plugging Together a Term

A plug can be inserted into a socket in one of two ways. Firstly, an unconnected structure (i.e. a sub-term of arbitrary completeness) can be picked up by its root icon and moved until it is over the desired socket. If the socket will accept this plug (i.e. they are type compatible) then the socket's image will flash to indicate that the insertion will succeed. The plug is then inserted by clicking a mouse button. An attempt to plug in a term that is not of the required shape will simply cause the plug to "fall off" the socket onto the view's background from where it can be picked up (if desired) and moved to a more receptive socket. This physical analogy is intended to reinforce the user's conceptual model of shaped plugs and sockets.

The second way of plugging in a new term is to place the cursor over the socket and issue the ADD SUBTERM command. This will pop-up a menu of only those operations that can be legally plugged into this sort of socket. When the user selects one of these operations, the corresponding term template is created and automatically plugged into the socket. An operation's *term template* is simply a graphical structure composed of an icon for the operator and sockets for each of the arguments required by this operation. This template is created directly from the operation's signature and is basically a tail-less version of the representation used in the Signature View. For example, the template for the `push` operation with signature `push: data × stack → stack` is shown in figure 6.4.

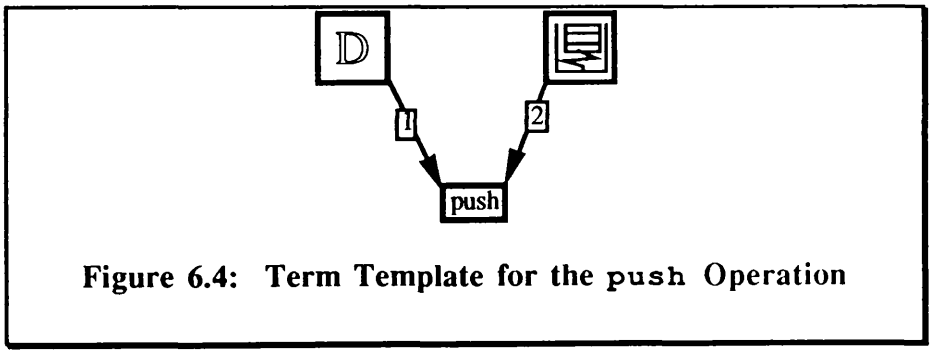


Figure 6.4: Term Template for the `push` Operation

With both of these approaches, plugging an object into a socket will *replace* the object that was originally plugged in. This features makes the graphical editing of terms a powerful way of quickly manipulating the tree-structures of terms and equations.

Consider starting to build equation 6.1 by adding the dominant term `is.empty?(...)`. This is achieved by issuing the ADD SUBTERM command and selecting the `is.empty?` operation from the menu of available operations. A term template for the operation is created which can be plugged in (either manually or automatically) to the left-hand socket shown in figure 6.3. Once plugged in, the Equation View will look like figure 6.5.

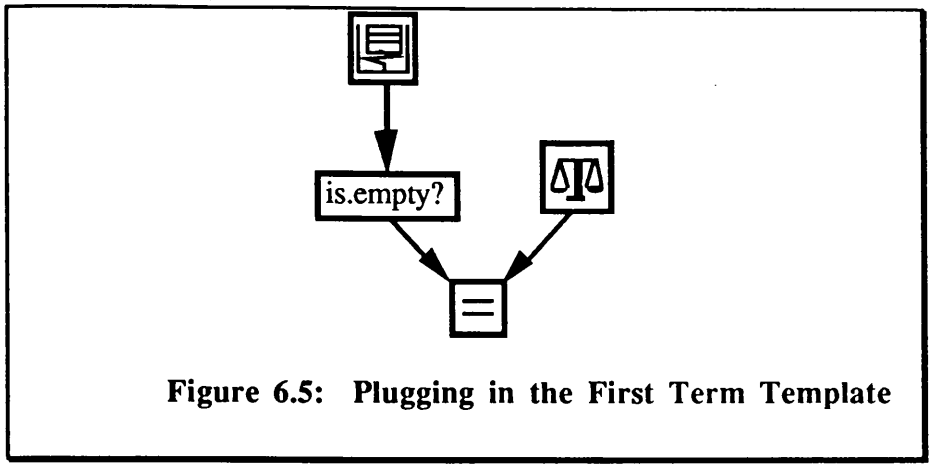


Figure 6.5: Plugging in the First Term Template

The newly inserted `is.empty?` template includes a new socket that will accept a stack shaped argument (as indicated by its stack image). Since the `is.empty?` operation produces a result of sort Boolean, the Equation View must ensure that the right-hand side produces the same value as dictated by the equation's equality constraint. This implies that the left and right-hand sides of the equation must produce values of the same sort and so it constrains the right-hand socket to accept templates only of sort Boolean in future. To indicate this change, the image of this socket changes from the original question mark to the image associated with sort Boolean.

Unplugging Terms

Since a term template automatically includes the links between the operation icon and its arguments, there is no need for an explicit command to add links to the graphical representation of a term. Links are implicitly created whenever an object is plugged into a socket: plugging an object T into a socket S will create a link $T \rightarrow S$. Similarly, links are broken by unplugging a term from a socket which is performed by placing the cursor over the socket and issuing the UNPLUG command. As an example, consider the situation shown in figure 6.6: (a) shows the situation before unplugging the `push` term from the argument socket of the `is.empty?` operation, with (b) showing the after-effect. The unplugged template is left lying on the surface of the view ready for further manipulation, if desired.

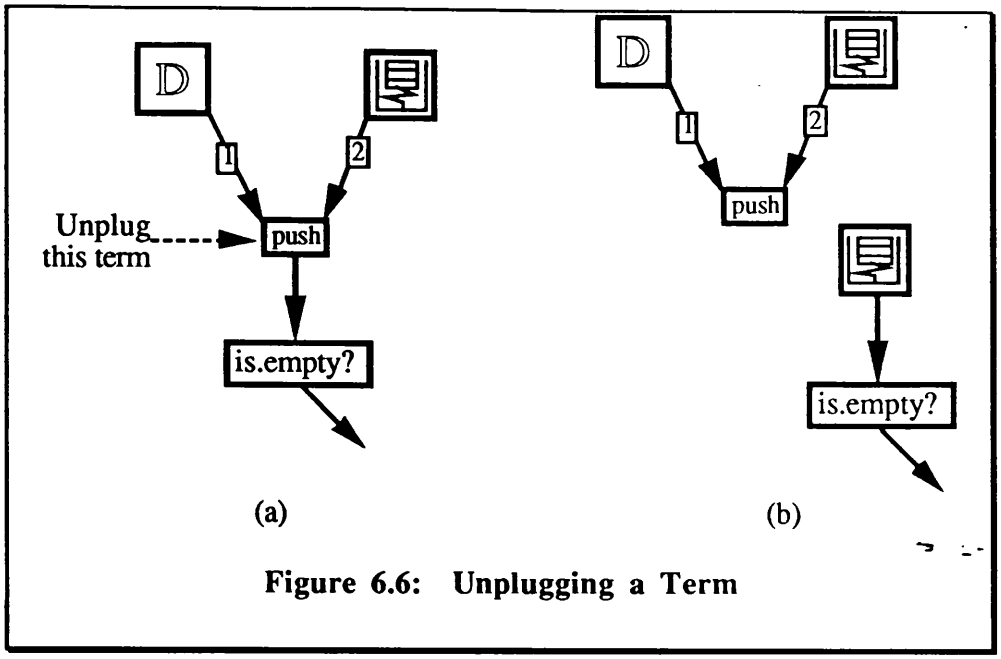


Figure 6.6: Unplugging a Term

Editing of Terms as Atomic Entities

When using the Equation View the user is dealing conceptually with objects such as terms and equations, and a good user interface will allow the user to edit these objects directly. Although the primitive editing commands to add and delete icons and links are appropriate for the Hierarchy and Signatures Views, the Equation View (and its relations) must provide higher-level editing facilities for the manipulation of terms as atomic units. To this end, VISAGE provides facilities to copy, delete and move entire terms, each via a single command. The user simply positions the cursor over the icon representing the root of the desired term and selects the appropriate command from the menu. Since these commands operate on potentially very large structures, any drastic actions (e.g. deleting an entire term) will require confirmation from the user before being performed.

The ability to operate on graphical structures corresponding to the conceptual units involved in the specification makes this form of editing much more natural and intuitive than the equivalent textual mechanisms, even when tools such as syntax-directed editors are available. Since objects can be referenced by pointing and then manipulated using the minimum of physical action (e.g. mouse button clicking or key presses), the graphical editing techniques developed for the Equation View are much more efficient in terms of the user's time.

An Editing Workbench for Terms

The Equation View can be regarded as a "workbench" where the user builds and modifies equations for the currently selected ADT specification. Although the workbench allows the manipulation of only one equation at any one time, the user may work on several different sub-terms of the equation independently, rearranging them into a form where they

can be plugged into the main equation, or just discarded. These parts are formed either via menu commands that create new term templates, or by unplugging or copying parts of existing terms. Only when the user issues the ACCEPT command is the current state of the equation incorporated into the specification.

Although the Equation View allows the user to leave isolated terms lying around on the surface of the view, they are transient structures that disappear when the user issues the ACCEPT command to freeze the state of the equation. There are times, however, when the user may want to use these components in several different equations or situations. For this reason VISAGE provides graphical *scrapbooks* each of which can store arbitrary numbers of terms in any state of completeness. These scrapbooks are explicitly created by the user and appear in their own window on the screen allowing them to be independently closed, collapsed, moved and resized. In VISAGE, the user can create an arbitrary number of scrapbooks thereby allowing related components to be grouped together for easy reference and access.

VISAGE scrapbooks are an extension of the facility provided in the Macintosh computer, (Williams, 1984). A scrapbook is used as a medium-term store for frequently used components: in VISAGE these components are terms that can be used to build equations or to test the specification (as discussed in a following section on the PLAYPEN). Terms are transferred between a scrapbook and the main VISAGE views via a special storage area which can hold at most one term. This is similar to the Macintosh *Clipboard.* The user can copy terms into and out of this clipboard using dedicated menu commands. *Any* view, whether textual or graphical, can be used as the source or destination for copy operations involving the clipboard; in particular, these include scrapbooks, the Equation and Specification Views, and the PLAYPEN. This means, for example, that a term may be copied from a textual representation, graphically edited in the Equation View and then copied out to a scrapbook for safe-keeping, or any other combination.

VISAGE scrapbooks extend the functionality of the Macintosh facility by allowing the user to select parts of terms if desired rather than being constrained to deal with whole terms. Using identical facilities to those provided in the Equation View, the user can select sub-terms simply by pointing at the icon at its root. This facility makes scrapbooks much more useful as repositories of parts likely to be useful in building specifications.

Animating an Equation†

The Equation View represents an equation as a tree rooted at an equality constraint. This tree can be regarded as a pair of dataflow trees, one each for the left and right-hand terms, that must produce identical final values. Values "flow" down from the leaves of the tree, get transformed at the operation nodes and eventually reach the root.

To help reinforce this dataflow concept, an experimental enhancement was added to the Equation View to represent the flow of values using simple animation techniques. As the values flowed through the tree, a "blob" would move along the corresponding links. To help indicate the sort of the values, these blobs would actually be miniature versions of the appropriate sort's image. The synchronisation of these blobs' movement mimics the dataflow convention of operations *firing* only when values are present on all its argument links. When an operation fires it consumes its arguments, shown by blobs moving in parallel along their links, and places its result on its output link. The values that exist on the links are terms that have been simplified using the equations of the specification as rewrite rules (discussed in detail at the end of this chapter). A value placed on a link persists until overwritten by a later animation. Once the animation has finished, the user can inspect the value that exists on an individual link using a special *Term Viewer*. This viewer provides a read-only, graphical display of the term in a similar format to that used in the Equation View itself. The user can open as many viewers as desired, giving the opportunity to compare before and after images of how values change as they flow through the equation tree.

Since VISAGE permits the use of conditional terms in equations, the animation facility had to be able to display the flow of values through the condition with an appropriate representation of its state i.e. whether its guard evaluated to TRUE or FALSE. To depict the switching nature of the conditional, the image of the icon in the conditional template can dynamically toggle between UNKNOWN, TRUE and FALSE states. These are shown (respectively) in figure 6.7.

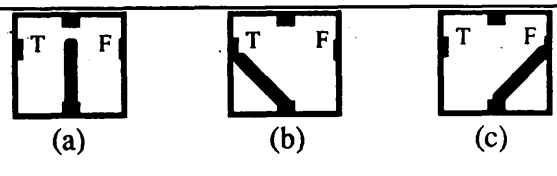


Figure 6.7: Animated Images of the Conditional Template

† Here the word "animating" is overloaded to mean both execution as well as the conventional one of a moving graphical display.

Before animation begins, the icon image is as in figure (a). Once the guard value has flowed in at the top of the conditional, the gate of the switch is moved according to the arrived value: a TRUE value will cause it to swing to the left (as in figure (b)), and a FALSE value will swing it to the right (figure (a)). This switching action promotes the idea of the conditional acting as a valve that lets through the value produced by at most one of its terms onto its output link. If the guard value could not be reduced to a simple Boolean value (because the set of equations is incomplete, for example) then the switch will remain in the neutral position as shown in figure (a), preventing the flow of values through the conditional.

Although the ability to animate values flowing through an equation was attractive and useful in helping novices understand the flow of data within an equation it had one major limitation that was particularly apparent when animating equations involving conditional terms. In order to demonstrate the switching action of the condition, the user must be able to assign different values to variables that appear in the guard term. For example, in the equation†:

```
includes? (x, push(d,s))  = if equal?(x,d)
                                then true
                                else includes? (x, s)
```

the user will want to assign different values to the variables `x`, `d` and `s` in order to understand how the conditional behaves. As a work-around solution, a menu command was provided to the Equation View to let the user assign simple terms to variables. These terms were selected from a menu and consisted of templates built from constructor operations of the appropriate ADT. For example, the variable `s` above is of sort stack which gives rise to the following terms:

1. `new.stack`
2. `push ('v1,'v2)`

Being built from the ADT's constructors means that together they can represent all values of their sort. Their generality derives from the use of system-generated variables to stand for arbitrary sub-terms: shown as `'v1` and `'v2` in the `push` term above. Unfortunately, although these terms did allow the user to exercise the equation using different values, their abstract nature diminished their usefulness in showing how the equation behaved. Users expressed a desire to use terms that they had constructed rather than choosing a seemingly arbitrary term created by the system. The pursuit of this goal lead to the development of the PLAYPEN (to be described shortly).

---

† The equation `includes?(new.stack) = false` is needed to fully define the `includes?` operation.

6.1.7 Editing in the Operation Dependency View?

As described previously, the Dependency View shows the relationships existing between operations resulting from the "calling" structure of all the equations in a specification. Given that the dependency relationships are changed by editing the structure of the equations (using the facilities provided in the Equation and Specification Views), is there any need for an editing facility in the Dependency View? What would editing mean here?

The only sensible editing facility that the Dependency View could support is the creation of partially complete equations. A simple rule-of-thumb for creating specifications is to identify the constructor operations and then define how the other operations behave when applied to each constructor. To illustrate this mechanism, consider the specification of the stack ADT given in Appendix C. This specification involves the two constructors `push` and `new.stack` with the three operations `top`, `pop` and `is.empty?` defined in terms of them.

To give an idea of the intended mechanism in action, consider the stack specification with no equations yet defined. The Dependency View could display icons for all the operations whose signatures were introduced in this specification; this is shown in figure 6.8.

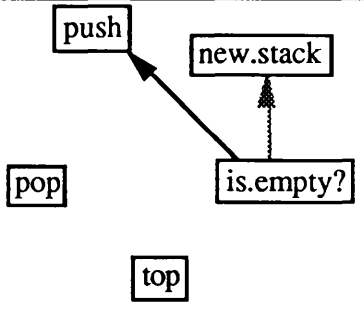


Figure 6.8: Adding Dependencies between Operations

The user now adds a dependency relationship between `is.empty?` and the constructor `push` by drawing a directed link between the two icons (shown as the solid arrow in the figure†). VISAGE would then automatically create the first template below:

```
1.  is.empty? (push (unknown!, unknown!)) = unknown!
2.  is.empty? (new.stack) = unknown!
```

Similarly, adding a dependency link from `is.empty?` to `new.stack` (shown by the gray arrow in figure 6.8) would give rise to template two above. The user could proceed in

† Note that the node-and-arc representation used in this figure is merely for illustrative purposes.

this manner until all the combinations had been addressed. The resultant equations could then be completed (by refining the `unknown!` terms) using the Equation View.

When developing a specification a common slip is to forget to specify the behaviour of an operation in a particular situation, e.g. the effect of applying `top` to a `new.stack`. A useful facility would have the user identify the constructor operations (perhaps by graphical selection) allowing the system to check automatically that all the combinations of applying operations to the constructors had been specified. Any omissions would then be reported to the user for consideration. Although such a facility is not currently present in VISAGE, the Operation Dependency View could be extended to include it, perhaps using a variant of Thiel's algorithm (Thiel 1983).

6.2 The VISAGE PLAYPEN

Since the behaviour of an ADT is determined by the actions of each of its associated operations, it is essential that a mechanism is available that allows the behaviour of each operation to be understood if one is to understand the whole ADT. An obvious approach to understanding an operation is to apply it to some example data. One possible way of depicting the dynamic properties of the operations is to use graphical animation techniques to show how data values change under the influence of various operations.

In chapter three, considerable discussion was focused on the use of examples in the programming process as a way of understanding a program's behaviour by watching it operate on a worked example. Users find it easier to understand the workings of a program if they can follow it as it operates on a concrete example of data rather than trying to grasp its behaviour from an abstract description. Since an executable specification is simply a program described at a high level of abstraction, a *specification-with-example* facility would offer the same benefits within a specification environment.

Observing the behaviour of operations implies that the operations of an ADT are executable despite only being defined implicitly in the algebraic specification. As chapter two described, such specifications can be made executable using term rewriting as their operational semantics. A specification-with-examples facility would allow the user to supply a term and have it rewritten into a simpler, equivalent term. This facility is provided within the VISAGE system by the PLAYPEN.

The PLAYPEN appears in a separate Smalltalk window allowing it to be opened, closed, resized and moved independently of the main VISAGE browser.

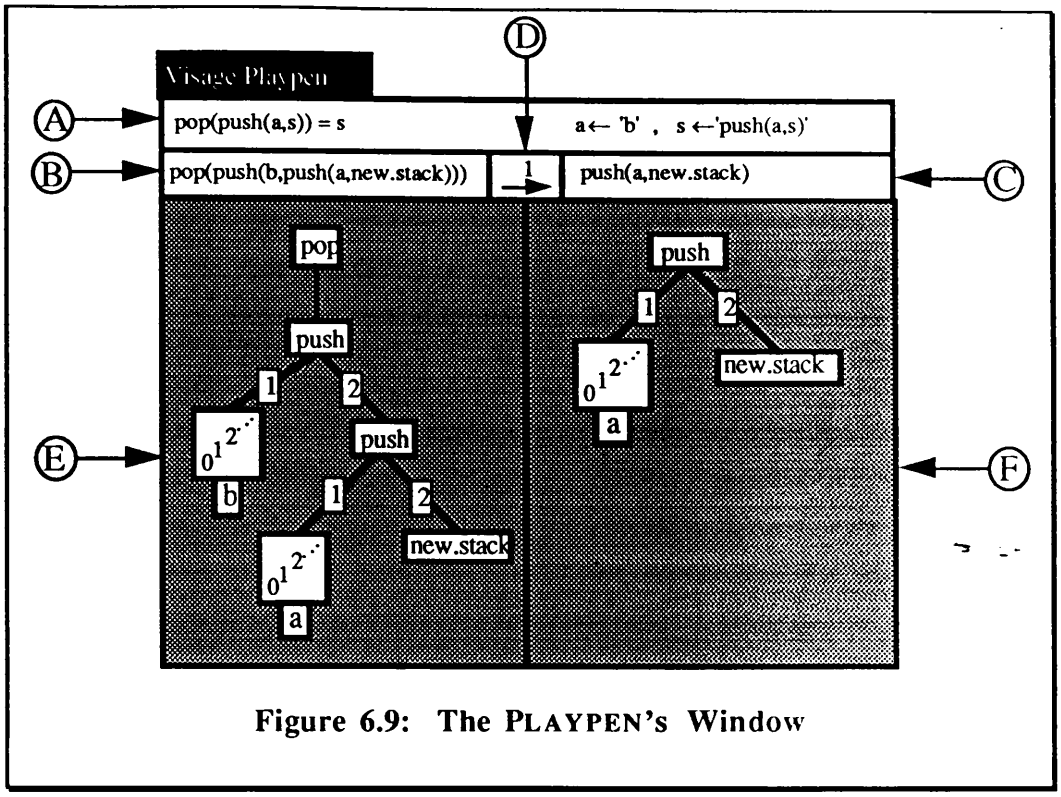


Figure 6.9: The PLAYPEN's Window

As figure 6.9 shows, the PLAYPEN window is composed of several different views. The window is dominated by two identical graphical views (indicated as E and F) that show a tree-structured graphical representation of terms. The user can construct terms in either of these views using the graphical editing commands developed in the Equation View. Above each graphical view is a textual view that shows the textual form of the term displayed graphically. The user can enter a new term by textually editing the contents of these views. For example, view B is the textual form of the term displayed in view E. Each pair of textual and graphical views are synchronised to ensure that any editing in one view is automatically reflected in the other. Alternatively, terms may be pasted into any of these views from the Clipboard.

When the user has created a term (either textually or graphically using the editing protocol developed for the Equation View) on one side of the PLAYPEN, that term can be rewritten into a resultant term that is then displayed in the opposite pair of views. In figure 6.9, the term created on the left-hand side has been rewritten and the result displayed on the right-hand side. The direction of the rewriting together with a count of the number of rewriting steps used is displayed in view D. The equation used to rewrite the term, together with any assignments to variables, is displayed in the view marked A. The resultant term can be edited if desired or can itself be rewritten. The user can therefore single-step through a sequence of rewriting steps watching how equations are applied at each step and how they rewrite the term.

6.2.1 Playing with User-Created Examples

As mentioned at the end of the discussion of the Equation View, the PLAYPEN was developed to meet users' desires to experiment with concrete example data that *they* had created rather than have to choose from a limited set of abstract, system-supplied examples. These example terms can be created using any of the techniques already discussed, e.g. graphical or textual editing, or pasting a previously saved term from a scrapbook.

As with the Equation View, these terms can be in any state of completeness. The PLAYPEN will always try to rewrite as much of a term as possible, regardless of how much has been omitted. This means that users can build a term to test a particular aspect of an ADT's behaviour without having to bother with irrelevant sub-terms. For example, in figure 6.10, an experiment is being conducted to check that pushing a value onto an arbitrary stack results in a non-empty stack. For this experiment it is not necessary to specify the value being pushed or the initial stack. These values are left as `unknown!` and are represented by icons whose image reflects their sort. Indeed, insisting that these sub-terms be added would obscure the fundamental point of the exercise. In some circumstances, supplying superfluous sub-terms can drastically affect execution performance as the PLAYPEN needlessly rewrites them.

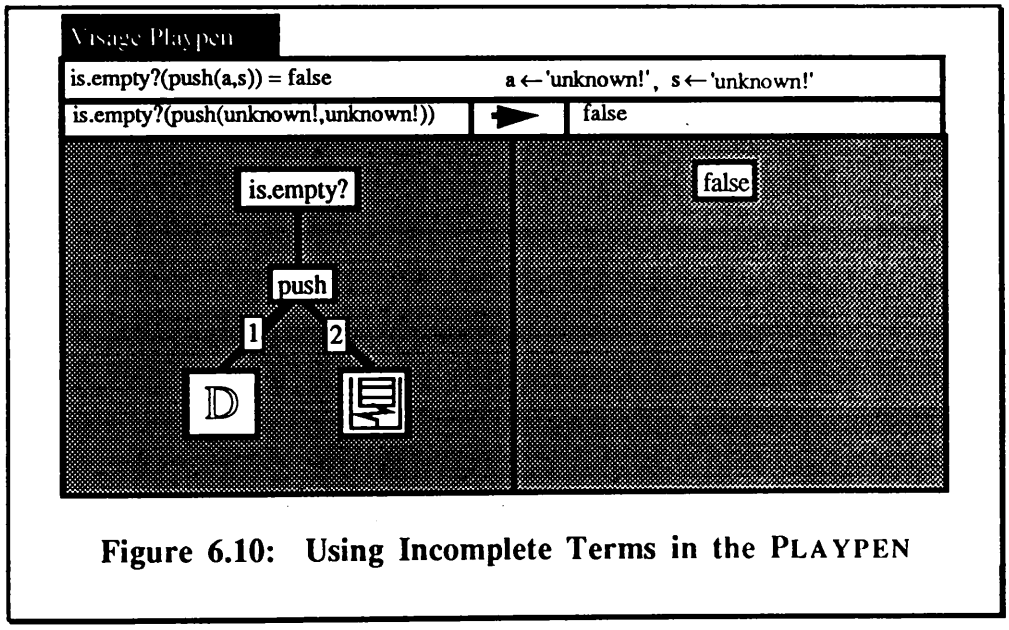


Figure 6.10: Using Incomplete Terms in the PLAYPEN

When experimenting with the PLAYPEN it is likely that the user will want to supply specific examples before being convinced that a particular operation is behaving as expected. Such terms will typically involve the constants of the appropriate sort. In addition, rather than leave sub-terms unspecified, the user may create and name variables to stand for arbitrary structures. These variables have a scope and life-span determined by the PLAYPEN in which they were defined.

6.2.2 Simplification Algorithm

The heart of the PLAYPEN is the term rewriting mechanism that reduces a term using the equations of the specification as left-to-right rewrite rules. This section describes how the basic term rewriting mechanism described in chapter two is implemented within the PLAYPEN.

For a given specification, let E be the union of the all the equations in this specification and all those that it uses. The rewriting mechanism takes a term (called the *target)* and tries to match it against one of the left-hand sides in E. The left-hand term of an equation is called the *pattern.* If a match is found then the corresponding right-hand side will be returned.

Given the two terms, target and pattern, the matching procedure works as follows. The two term trees are searched simultaneously in a depth-first manner until two nodes are found with different operators: call the target operator θ_T and the pattern operator θ_P. If θ_P is not a variable then the match procedure fails. If θ_P is a variable then this difference in the terms can be resolved by substituting the sub-term starting at the node θ_T for the variable θ_P throughout the pattern term so that the variable cannot be matched later to a different sub-term. This substitution will be included in the final output of the match procedure. Once this substitution has been made, the search for differences continues until the terms have been completely searched. If no irreconcilable differences remain at the end then the target and pattern terms match.

To help understand the matching procedure, consider the following example. The target term `pop(push(0,push(1,new.stack)))` is to be matched against the equations of the stack ADT. Consider two candidate equations for the match:

```
1.  pop(new.stack) = new.stack
2.  pop(push(d,s)) = s     "Where d and s are variables"
```

The first equation's left-hand side is used as the pattern in the following tree search.

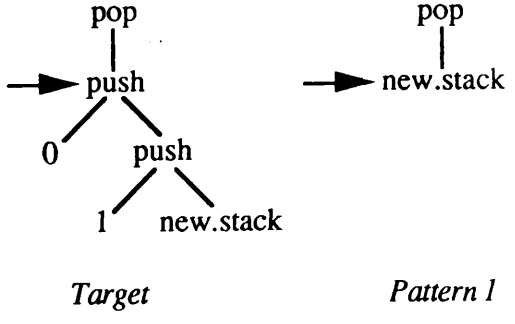


Figure 6.11: An Unsuccessful Match of Terms

The nodes marked with arrows denote the first difference found. Since the `new.stack` operator is not a variable no substitution will reconcile this difference and so the match fails at this point. The second equation is then used to provide the pattern. Its search is shown in figure 6.11.

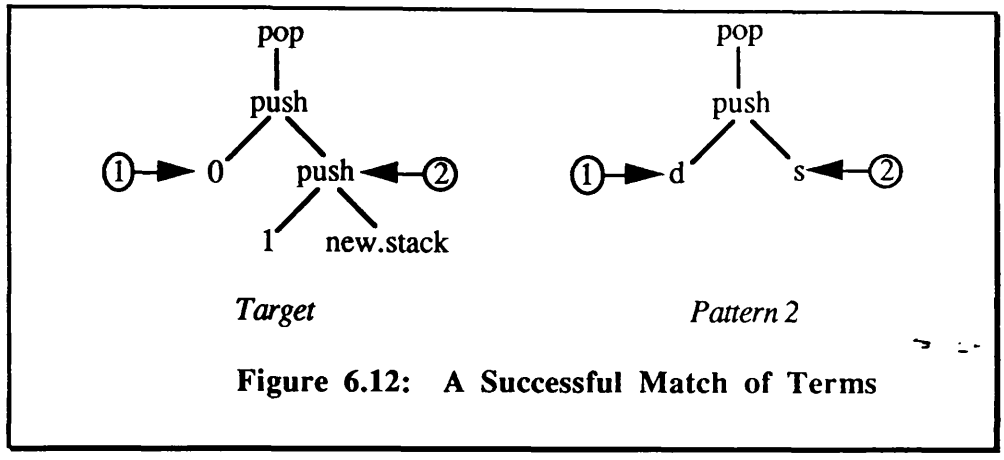


Figure 6.12: A Successful Match of Terms

With this second pattern the match procedure found two differences as marked in figure 6.12. Since in both cases the pattern operator is a variable, these differences can be reconciled by the following substitutions: `d ← 0, s ← push(1,new.stack)`. The match procedure therefore succeeds and returns the above substitutions as its result.

Once a match has been found, any substitutions required to make the match must be applied to the right-hand side of the matching equation. In the above example, the right-hand side is simply the variable s and so after the substitution, the result of the rewriting operation is `push(1,new.stack)`.

6.2.3 Evaluation Strategy

The match procedure described above tries to match a target term with some candidate equations. However, as discussed in chapter two, there are several strategies for reducing a term to its canonical form. In general, the most suitable strategy is the so-called Outside-In Application as this tends to produce the canonical form much faster than other approaches. To recap, this strategy first tries to rewrite the entire term but if this fails, it recursively tries to apply the rewriting mechanism to the sub-terms of the original until one is found to match. If no sub-term can be matched against an equation then the overall term is irreducible and the rewriting process stops.

Due to its relative efficiency in rewriting a term, the outside-in approach was used in the PLAYPEN. A future development of the PLAYPEN would provide different evaluation strategies, allowing the user to compare the differences between the alternatives. The PLAYPEN display highlights (both textually and graphically) the sub-tree of the overall term that was rewritten, together with the equation used to rewrite the term along with any substitutions required by the match procedure. The visualisation of the rewriting

mechanism could be easily improved in future versions of the PLAYPEN by means of an animated display of the search for matching equations and a trace of the evaluation strategy.

6.2.4 Single Stepping and Continuous Rewriting

The basic rewriting mechanism converts an input term into a resultant term by the application of an equation: denoted as $T_1 \rightarrow T_2$. The resultant term T_2 may, in its turn, be rewritten by some equation into a further term. This application of successive equations is called *single stepping*. At each step in the process, the user is able to inspect the before and after terms together with the equation that was used to rewrite one into the other. If desired, the user can continue single stepping through the rewriting sequence until the point is reached where no equation matches the term: this term is said to be *irreducible*.

Although single stepping is useful in following the rewriting process, it is often desirable to miss out displaying the intermediate terms and to continue rewriting until an irreducible term is found. If the rewriting sequence is represented as $T_1 \rightarrow T_2 \ldots \rightarrow T_n$ where T_n is the irreducible term, then the PLAYPEN will only display the start term T_1 and the resultant term T_n. The term T_1 can be thought of as input to an equationally specified program, with T_n as the output. To provide some indication of the progress of the rewriting process, the PLAYPEN updates a counter on the screen after every rewriting action.

As discussed in chapter two, there is the possibility for the rewriting process to enter into an infinite loop. In a teaching environment, such a situation would be disconcerting and therefore should be trapped by the system. Unfortunately the nature of the Halting Problem prevents a perfect test for a non-terminating rewriting sequence and so heuristic checks have to be applied. Being heuristic, these checks will not be perfect in trapping run-away rewriting. However, by ensuring their test criteria verge on the pessimistic, these checks should always trap non-termination at the expense of occasional false alarms.

The PLAYPEN uses two checks for trapping an infinite rewriting sequence. Firstly, a count is kept of the number of rewriting actions applied so far and if this exceeds an implementation limit then the user is asked whether the process should be terminated. The PLAYPEN also displays the rewriting count on the screen and if the user feels it is getting too large, can manually terminate the process. The second check done by the PLAYPEN is to monitor the size of the terms being generated by the rewriting process. In a well-behaved rewriting process, these terms will get monotonically smaller. If they start to get larger then the rewriting process is diverging and should be terminated. However, with some terms it often the case that terms must get larger before they get smaller and so a strict monotonicity check is insufficient. The PLAYPEN therefore looks at the long-term trend in term size and only terminates the rewriting process if the *trend* is for terms to get larger. Whenever the rewriting process is terminated, the latest generated term is always

displayed so that the user can see what has happened, together with the reason for the stoppage.

6.3 Summary

This chapter described the extensions of the basic visualisation machinery developed in chapter five to allow ADT specifications to be created and modified by graphical means. The PLAYPEN, an animated, graphical term rewriting environment, was also described. With these extensions VISAGE is now capable of displaying, editing and testing ADT specifications within a unified, graphical environment. The following chapter evaluates the success of this prototype environment by analysing the performance and reaction of users doing simple specification tasks both with and without VISAGE.

Chapter Seven
Evaluation of VISAGE

"I observe that the uninformed and untested intuition of the designer is almost always wrong. We must always refine our interfaces by tests with real users, and we will always be surprised by those tests. ... Or, put simplistically, Any data are better than none."

Brooks (1988, p.2)

7.1 Introduction

The VISAGE system was intended to allow people unfamiliar with the formal specification of ADTs to gain an intuitive understanding of their form and behaviour. To determine whether this goal has been reached required an evaluation that gets real users to use the system on real tasks and monitoring their performance: *"...methodological recommendations of computer science should be recognised as* empirically testable, psychological hypotheses", Sheil (1981, p.101).

This chapter presents the development of the evaluation method, an analysis of the results obtained, and a discussion of the limitations of such an evaluation of the VISAGE system. It includes an examination of the issues and problems raised by the evaluation and the effect these have on the implementation.

The main purpose of an evaluation such as this is not to prove that the system is "good" but instead to find out what is wrong with its design, to probe users' attitudes to using the system, and to reveal strengths and weaknesses in the interface. A poor result is an indication to the designer that something needs to be improved. The evaluation is not the final stage in building a system but instead the start of the next design or implementation iteration. The results of the evaluation should be used to improve the design which in turn is evaluated, and so on until an acceptable version remains. This chapter presents the results of such an evaluation and in the light of these results, discusses future refinements and developments of the VISAGE system.

The aim of the investigation was to discover the extent to which VISAGE assisted in developing algebraic specifications of ADTs in a graphical manner and in ensuring that these specifications match the expected behaviour.

7.2 Pilot Study

As with any kind of design, it is highly unlikely that the best evaluation strategy will be found at the first attempt. The design of the evaluation itself requires several iterations of

refinement before satisfactory materials and method are found. It is the job of a pilot study to identify any problems in the evaluation and to recommend improvements before it is used with the intended subjects.

The pilot study of the VISAGE evaluation used a specialist in evaluation techniques to follow the route to be taken by the subjects and to find as many problems as possible in the evaluation and the system itself. The advantage of using such a specialist is that as well as identifying problems, he can also suggest possible improvements and changes. The results obtained from the *expert evaluator* were confirmed by comments from a second user with knowledge about VISAGE itself who checked the evaluation from the system viewpoint.

Each pilot run took over three hours and identified several parts of the evaluation material that needed improvement. These included rectifying badly-phrased sentences, vague questions and omissions as well as glaring problems with the software itself. An important improvement concerned the detailed description in the tutorial of how to build equations graphically. The original technical description was replaced by a metaphorical one intended to appeal to the subject's intuitions.

7.3 Evaluation Method

7.3.1 Introduction

Once the results of the pilot study had been incorporated into the evaluation material, a larger group of subjects could be used to evaluate VISAGE itself. This section describes the method used to perform that evaluation. A brief description is given of each phase of the evaluation in the following sub-sections.

7.3.2 Overview of the Evaluation

Pre-Test

Each evaluation session took place with a single subject in a quiet room to ensure the minimum of disturbance and took about two hours to complete. Each subject used exactly the same materials and version of VISAGE, thus standardising the evaluation. The session started with the subject being told the purpose of the evaluation, emphasising that it was VISAGE being tested and not them. This ensured that the atmosphere during the session would be relaxed and informal and so conducive to a successful evaluation.

The subject was then asked to complete the first of two questionnaires (included as Appendix E of this thesis). This has five questions and provides estimates of a subject's experience of using a workstation and the Smalltalk system, and their confidence in writing and understanding the specifications of simple ADTs prior to using VISAGE.

Learning Phase

The next phase introduced subjects to the concepts and functionality of VISAGE. This was standardised by having subjects go through a tutorial that illustrated a subset of the available commands through the hands-on development of a simple example specification. During this phase the subjects were asked to articulate their thoughts about any problems they were having, and their goals and feelings to using the system: a so-called *think-aloud protocol.* Informal notes were taken of each subject's reactions and comments for later analysis. There was no time limit for this phase but subjects took 63 minutes on average to complete it (with a standard deviation of 11 minutes).

Task

The next phase of the evaluation was a five-part exercise to extend and modify an existing specification using the techniques acquired in the learning phase. The exercise description is included as Appendix F. This phase was timed and the subject's activity monitored for later analysis. The subject was always free to refer to the tutorial if they got stuck, but were encouraged to verbalise their difficulty as this often helped them overcome it. These comments were an important instrument for determining how subjects felt about using various aspects of the system. The exercise was carried out immediately after the tutorial introduction and so before the operational knowledge picked up in following the tutorial had faded in the subjects' memory. The specification produced during the exercise was saved in a file for later assessment.

Post-Test

Immediately after finishing the exercise, the subject was asked to complete the second questionnaire that consists of twenty questions (included as Appendix G of this thesis). This questionnaire was used to gauge the subject's reaction to using VISAGE along various dimensions, and to detect any change in confidence or attitude in dealing with formal specifications of ADTs. At the end of the questionnaire the subjects are invited to write comments about what they thought were the best and worst aspects of VISAGE: this gave them the freedom to comment in their own words on aspects not covered by other questions or to elaborate on previous responses. When the questionnaire had been completed, the responses were discussed with the subject to remove any confusion and note any comments the subject may have made e.g. regarding possible improvements to VISAGE. This completed the evaluation session.

7.3.3 Environment

The same workstation† was used with each subject to prevent variations in computer response time from interfering with the evaluation. The subject always manipulated the keyboard and mouse apart from the steps to start and stop VISAGE execution, and handling the display of windows, since these actions are not covered in the tutorial. The evaluator sat beside the subject and took informal notes of any comments made by the subject, and also descriptions of any problems encountered.

7.3.4 Subjects

Ten subjects took part in the evaluation of VISAGE, compromising seven post-graduates, two post-doctorate and one undergraduate. All subjects had reasonable experience of using computers but they did not all have experience of programming. Their technical background was mixed and included interest or expertise in human-computer interaction, formal methods and functional programming, among others.

7.3.5 Control Group

The results obtained by the evaluation provide a qualitative measure of how the subjects' felt about using VISAGE to create and modify ADT specifications. In order to put these responses in perspective, a separate, *control group* of subjects were asked to perform the same task as those in the main group but using paper and pencil rather than VISAGE. They were asked to complete questionnaires similar to those used for the main group but without reference to VISAGE. The subjects in the control group were given an equal amount of tuition of the algebraic specification of ADTs as those in the main group.

The control group in the evaluation comprised four subjects, one post-graduate familiar with algebraic specifications, and three undergraduates with no experience of such formal methods. The subjects in the control group all had similar computing experience to those in the main evaluation group.

7.3.6 Questionnaires

Questionnaires were used as an evaluation instrument because they allow reasonably accurate responses to be gathered quickly and cheaply while the subject still remembers the feelings of using the system. Although the responses are of a subjective nature, the general trend in the results can provide a reasonable summary of the reactions of the subjects.

---

† A Sun-3/140 workstation with eight megabytes of main memory connected to a remote file server.

Each questionnaire has a set of simple, precise questions about a particular aspect of VISAGE and its user interface. Associated with each question is a response scale with either five or seven points. These limits were chosen since increasing the number of points on the scale beyond about seven has been shown not to improve the accuracy of the results; GUCHCI (1988, p.17). The questions with five-point scales were used to determine previous experience and each point has a sentence that typifies the level of experience. The subject circles the point nearest to their own experience. A five-point scale is enough for this class of question as only a rough classification is required. The majority of questions have numbered, seven-point scales with *anchor points* at each end. An anchor point is a verbal description that fixes a defined response on the scale. For example, a question about the effort required in learning the system might have the anchor points *Difficult* and *Easy*. The subject compares their own reaction against these anchors and choses a response relative to them. Since it is not always possible to capture neatly a reaction with a simple question, subjects were encouraged to qualify or elaborate on any answer by writing comments underneath their response.

7.3.7 Think-Aloud Protocols

Nielsen (1988, p.1) states that *"the think-aloud technique is one of the most important techniques for practical evaluation of user interfaces"*. The technique provides qualitative feedback about what it is like to use the system and ensures that all problems are recorded: asking a subject at the end of the session will probably identify only the major problems.

The subject was first assured that it is the *system* that was being evaluated and not them. While the subjects were using the system they were actively encouraged to articulate their thoughts about what they were trying to do, what problems they encountered (no matter how trivial) and what they liked or disliked about the system. These comments were recorded for later analysis. If a subject got stuck then the natural temptation to give help had to be resisted (lest the evaluation should suffer) so that their attempts at correcting the problem could be observed.

Think-aloud protocols were used to identify problems during both the pilot study and the actual evaluation of VISAGE. Since each evaluation took about two hours, subjects were not asked to articulate their every act but only when something bothered them or if they got stuck. This simplified the collection and analysis of responses.

7.3.8 Selecting an Exercise

To determine whether VISAGE is really of benefit it was necessary for each subject in the evaluation to perform some task using the system. To be credible, the task must be realistic, i.e. it must be something the subject would normally do, otherwise it would not provide any indication about how the system would benefit users doing real tasks. The

task must be sufficiently difficult that subjects have to think about its solution with the potential for getting into trouble, but must not be so hard that they cannot complete it in a reasonable time span. It is important that the task be the same for all subjects so that direct comparisons in their performance can be made. The task should not require any specialist knowledge other than that given in the evaluation itself or reasonably expected to be known by all subjects.

The chosen exercise requires the subject to extend a specification of stack: the exercise description given to subjects is included as Appendix F. The stack data type was chosen as it should be understood by all subjects, being an important and commonly-occurring data type. This was reinforced by having the exercise continue the example developed in the tutorial. To ensure that all subjects started from the same point, a specification of stack was provided in a separate file.

The evaluation exercise has several steps that try to test the subject's use of the different techniques introduced in the tutorial. The first step asks the subject to add a new operation to the stack specification whose behaviour is described informally. This tests their ability to select screen entities, create a new icon representing an operation, and add links between icons to define the signature of the new operation. The second step asks them to give a comment to the newly created operation, which tests general text editing and use of the VISAGE help facility. The third step asks them to create two equations that define the behaviour of this new operation, using the graphical editing mechanisms. Once they have created the equations they are asked to use the PLAYPEN to ensure that the operation behaves as they would expect.

The fourth step in the exercise asks subjects to guess the behaviour of an operation called MYSTERY (it actually concatenates two stacks together). This operation was chosen because it is not one normally encountered with stacks and so subjects would not be able to guess its function easily. The first difficulty is that the name of the operation gives no clue to its function. As Guttag (1975, p.12) states: *"To rely on one's intuition about the meaning of names can be dangerous when dealing with familiar types. When dealing with unfamiliar types it is impossible"*. In addition, most people inexperienced with the recursive equations defining the behaviour of this operation would find them difficult to understand textually, and hopefully this will encourage them to use the PLAYPEN to guess what it does. Once they had confirmed the function of MYSTERY using the PLAYPEN they had to rename the operation to something more descriptive, e.g. CONCAT. The RENAME command they had to use is not described in the tutorial and so this was a check of the subject's understanding of the use of each view and menu, as well as their confidence in exploring the system. The final step of the exercise asks them to save the modified specification to a file.

7.3.9 Tutorial Introduction

As Sheil (1981, p.103) points out: *"Completely novel environments ... require that the programmers being studied be thoroughly trained for their new environment, lest learning transients dominate the results"*. To ensure that all subjects were equally and sufficiently exposed to VISAGE before attempting the evaluation exercise, a 20-page tutorial on the system was prepared and used as part of the evaluation. The tutorial introduces the subject to the main concepts and mechanisms of VISAGE by stepping through a short example that specifies and tests a specification of a stack. The tutorial is included as Appendix D.

Carroll et al. (1988) highlighted the problems with traditional computer training material and proposed a radical alternative: the *Minimal Manual*. The VISAGE tutorial adopts some of the minimalist techniques they suggest. These include keeping the word count to a minimum and avoiding excessive use of jargon, in this case the use of mathematical terminology. In particular, the tutorial makes use of metaphors e.g. plugging terms into sockets of particular types, in an attempt to simplify the descriptions. VISAGE concepts and facilities are introduced in a user-oriented manner through the development of a simple example, instead of merely listing the system's functionality, since as Smith (1977, p.20) notes: *"From an educational stand-point, educators ... have long known that the concrete is easier to understand than the abstract"*. The tutorial has many (stylised) screen dumps of what should be on the screen for each step of the development to give feedback and reassurance to the subjects that they are progressing satisfactorily. Since the tutorial was designed in the knowledge that the evaluator would be present when the subject was using it, only basic error recovery techniques are described (which deviates from Carroll who emphasizes supplying error recovery information). This helps keep the tutorial to a manageable size. In the event, subjects encountered few errors that required external assistance. The problems in using the tutorial are discussed in the section below on *Learning the System.*

7.3.10 Measuring Task Performance

Once an evaluation exercise had been selected, it was necessary to decide how the subject's performance on the task was measured. The final step in the exercise was to save the resultant specification to a file. The obvious way of measuring performance is to compare this file with a model solution with marks being assigned for successful completion of the individual steps of the exercise. The marking scheme must be flexible enough to be tolerant of solutions that are different to the model but still correct. The exercise used in the evaluation was quite simple and tested the subject's use of only a subset of the available facilities. The results of the exercise could therefore only give a crude indication of performance, its main rôle was to confirm the subject's response in other parts of the evaluation, e.g. ability to construct equations graphically.

The exercise was marked with a maximum score of ten, broken down as follows. Two marks were given for creating the new operation, giving it the correct argument and result links, and supplying a suitable comment for the new operation. Six marks were given for creating two equations, one for the push case and one for the new.stack case. These equations had to be completely defined (with no unknown! sockets left) and properly oriented (for left-to-right rewriting). The last two marks were given for correctly discovering the functionality of the operation named MYSTERY and renaming it.

7.3.11 Monitoring User Actions

During an evaluation it is useful to know which commands the subject executed and the frequency of their use. This allows heavily-used commands that would benefit from optimisation to be identified. Conversely, those commands that were seldom used should be investigated to see why they were not used: are they really useful (if not, remove them); or, is the documentation describing them poor, for example.

For several reasons, a subject can only be able to provide an *estimate* of the command usage after using the system. Firstly, if the evaluation takes a long time then the subject will quickly forget which commands were used. Similarly, a delay in asking the subject will reduce the reliability of the answers. Secondly, the subject may not *know* the name of a command but only its function. For these reasons, the monitoring of user actions should be performed by the system itself. VISAGE can produce a summary report of a subject's actions into a log file for later analysis. This report includes an itemised list of all commands used and their frequency of use, ordered by decreasing frequency. It also records the frequency and class of any errors made by the subject. Finally, it gives the total number of different commands used, the total number of commands issued and the time taken for the session.

In this evaluation, each session log can provide evidence of the confidence of the subject in using the system. The tutorial introduction to VISAGE only describes a subset of the available commands; since all commands are available via menus, a confident subject will explore other commands to see what they do. This exploration was recorded in the subject's log and compared against the confidence rating as given in their questionnaire responses.

7.4 Results of the Evaluation

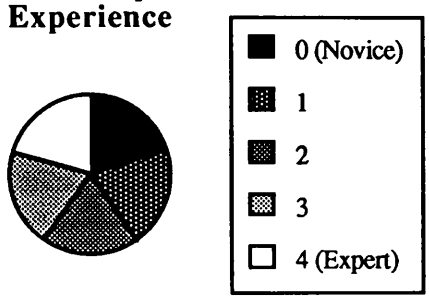
In this section, the results of the evaluation are presented and discussed in general terms. When appropriate, the average results (with standard deviations) of the questionnaires are also given to show the general trend in responses. (See the section on the limitations of the evaluation for a discussion on the interpretation of these values). The subjects' responses have been converted to percentage values with 0% indicating low or poor values

and 100% indicating high or good responses. Averages are shown by μ and standard deviations by σ. The average response shows what the subjects felt in general about an aspect, with the deviation showing how much in agreement they were, with a low value indicating high accord. Comments that subjects made during the evaluation are sometimes included with the results to illustrate a particular reaction, and are displayed in italic font.

7.4.1 Previous User Experience

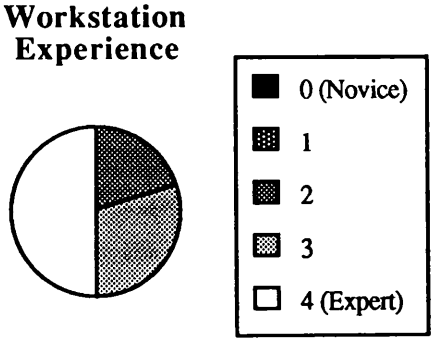
Before the subjects encountered the VISAGE tutorial or the evaluation exercise, they were asked to rate their experience of formal specifications, use of computer workstations and the Smalltalk-80 environment, on a five point scale ranging from novice to expert. The responses are shown in the following figures.

Formal Spec. Experience



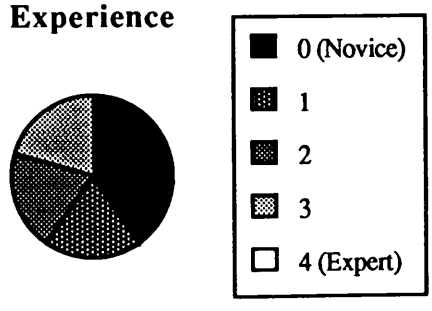
The responses show that experience with formal specifications is well distributed among the evaluation subjects. This allows sample reactions to be taken from a variety of different kinds of user to spot differences in their responses.

Workstation Experience



The high rating (with little variation) for workstation experience is predictable, as the subjects are mostly postgraduate students and staff in a computing science department.

Smalltalk Experience



Most subjects had little experience of using the Smalltalk-80 programming environment, with no subject regarding themselves as expert. This result is significant because the learning required to handle the Smalltalk interface will increase the effort required to become proficient in using VISAGE.

To minimise the learning effort, the evaluator always dealt with starting and stopping the VISAGE session, leaving the subject to deal only with the Smalltalk features required by

every application, namely the protocol associated with each mouse button, and the window scrolling mechanism.

7.4.2 Confidence in using Specifications

One of the main aims of the evaluation was to see whether subjects' confidence and facility in using algebraic specifications of ADTs increased through using VISAGE. As part of the pre-evaluation questionnaire, subjects were asked to rate their confidence in understanding an existing simple specification and in writing a new one. The subjects were asked to do this again immediately after using the system. Responses were on a seven-point scale with anchors *Not Very Confident* up to *Very Confident*. The results of this comparison are shown graphically in figure 7.1.

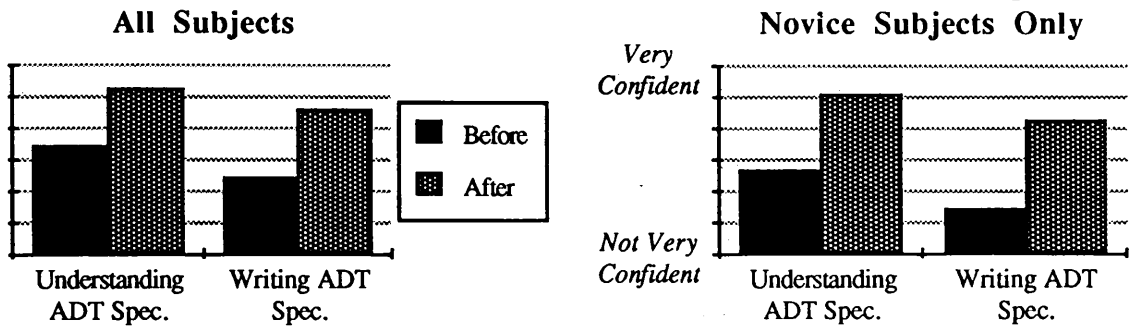


Figure 7.1: Confidence in using Specifications

Apart from two subjects who are expert in using formal specifications and gave maximum confidence ratings before and after using VISAGE, *all* subjects reported a marked increase in their confidence in both reading and writing simple specifications, as shown in the left-hand chart in the above figure. If these two expert subjects are discounted, the average difference in the before and after responses are even more dramatic, as shown in right-hand chart in figure 7.1.

Generally, subjects were more confident of their ability to read and understand an algebraic specification than their ability to write one. This is perhaps explained by the subjects being computer scientists who are used to dealing with a variety of formal languages and would be reasonably confident of guessing the behaviour of an ADT from its specification using this experience. However, *writing* in a formal language requires a detailed knowledge of the precise syntax and semantics involved that would make an inexperienced subject less confident writing a new specification from scratch. It is important to remember that using VISAGE, the *writing* would actually be done graphically.

7.4.3 Reactions to using VISAGE

Learning to use the System

VISAGE has been designed to let users quickly master its user interface allowing them to proceed as quickly as possible to learning about the formal specification of ADTs. This is possible by having a small, consistent set of commands. The evaluation shows that this goal has been reached because after only an hour's introduction on average, all subjects had mastered a working subset of VISAGE commands and were able to complete the exercise to a high standard with little reference to the tutorial. On a scale with anchors *Never* and *Constantly*, subjects' response for their need to refer to the tutorial during the exercise was μ=25%, σ=24%. As expected, subjects inexperienced in formal specifications needed more assistance and referred to the tutorial more often than more experienced subjects: this accounts for the high variation in their responses.

In many cases, if the subject got stuck they could often apply common sense in working out how to continue. An example of this is the step in the exercise where they have to rename an operation; something *not* covered in the tutorial. Although some subjects had previously noticed the RENAME command on the icon menu and applied it immediately, other subjects were stuck. If these subjects were asked to talk through the problem, they could often reason as follows: *"I need to change the name of an operation and the Signature View manipulates operations so I should look there. Now I need to change that icon. Well, each icon has a menu so there should be a command to rename it: oh yes! there it is"*.

Despite the attempt to write a tutorial that concentrated on introducing VISAGE from a user-centred, task-oriented approach, a few subjects had problems related to the tutorial. As Carroll et al. (1988, p.126) points out, *"New users are not inclined to read training material"*. This was illustrated by one subject ignoring the tutorial completely from the start confident in the belief that they knew how to use the system, only to quit the system through an inadvertent menu selection! When asked about this action, the subject said that their previous experience with direct manipulation interfaces, and in particular the Macintosh, gave them the (erroneous) belief that they could use *any* similar system despite there being major differences in the user interface. Unfortunately, there seems to be little way of *making* someone read training material if they believe they know what it contains.

When working through the tutorial several subjects skipped large procedural sections once they mastered the basic mechanism being described. The state of the display and the position in the tutorial then became unsynchronised. However, the inclusion of many screen dumps allowed the subject to re-establish synchrony.

Using the Mouse and Pop-up Menus

Since VISAGE has a direct manipulation interface, the mechanisms for selecting objects and issuing commands from a menu are particularly important. Consequently, the post-exercise questionnaire has two questions to solicit the subject's feelings about this aspect of the system. In addition, the subject's session log contains records of all selection mistakes and cancellations of commands issued by mistake.

Subjects were asked to rate their experiences of selecting objects using the mouse and in using the pop-up menus. Responses were on a seven-point scale with anchors *No Problems* and *Lots of Problems.*

In general, subjects found the selection mechanism straightforward and did not experience many problems (μ=13%, σ=11%). This is confirmed by the small number of errors recorded in the logs and is probably due to the similarity with the Macintosh interface with which most of the subjects are familiar.

The pop-up menus gave slightly more problems (μ=17%, σ=12%), the most basic being that subjects were confusing the rôle of the different mouse buttons: this quickly disappeared with increasing familiarity with the Smalltalk mouse convention. This problem was particularly noticeable in subjects with little experience of workstations equipped with multiple-button mice.

VISAGE menus are context-sensitive and the time taken to analyse the context of the cursor (i.e. whether it is over the background or a particular kind of icon) causes a slight delay in displaying the menu on the screen. If the delay becomes noticeable, subjects unfamiliar with the menus may think that something is wrong and release the mouse button. However, the menus remember their previous selection† and move the cursor to it automatically: releasing the mouse button causes that selection to be re-selected. Once subjects were aware of this problem they became more deliberate in their button pressing. This problem illustrates the need for trade-offs in user interface design. One subject thought that having the menus remember their previous selection was the worst feature of VISAGE, while another thought it was a good feature, being useful for repeated commands (such as adding links). An implementation compromise adopted in VISAGE is to have menus remember the previous selection but have a cancel mechanism for *dangerous* commands (such as DELETE) in case they are issued by accident.

---

† VISAGE menus are described in chapter four.

Graphical Construction of Signatures

Most subjects found the graphical construction of operation signatures to be straightforward, and after a couple of examples of adding links, quickly defined all the operations' signatures. The only problem was in adding links going in the wrong direction. The subjects who encountered this problem used two different techniques to correct their mistake. One group guessed the existence of the DELETE LINK command, searched for it (being pleased that common sense can help them use the system) then used it. The other group used a more brute-force approach and deleted the associated operation icon (having seen the DELETE command on the icon menu), which also deletes any attached links.

Different subjects had different styles of constructing signatures. When placing the icons for new operations, some subjects carefully selected a location that would minimise the length and number of crossings of the links that would be required; while others placed the icons at any vacant space and then cleaned up the diagram afterwards. When adding the links to define an operation's functionality, subjects usually added the argument links first and then the result, although some did it the other way round, or even in mixed order.

Graphical Construction of Equations and Terms

Several questions in the post-exercise questionnaire are devoted to finding how well the subject managed to construct equations and terms graphically. The tutorial introduction has a detailed step-by-step discussion of this mechanism since it will be novel to most subjects. The tutorial uses the metaphor of having plugs and sockets of various sorts, with the sockets (representing nodes in the equation tree) only accepting plugs (representing sub-terms) of the appropriate sort. All subjects found this metaphor to be useful (μ=75%, σ=22%) and many used the associated terminology when constructing equations e.g. *"I'll just plug this push into this argument socket here..."*, an indication that the metaphor is being used as their conceptual model. As might be expected, novice subjects found the metaphor to be of more benefit than the expert subjects. The subjects with previous experience of formal specifications stated that they only needed the metaphor to understand the mechanism involved and quickly developed their own conceptual model, usually involving the refinement of incomplete parts of the equation tree. This difference accounts for the large variation in the subjects' response.

Subjects found the graphical construction mechanism to be straightforward and had few problems in constructing equations and terms of reasonable complexity, after going through one or two examples; μ=79%, σ=21%. Once again, the large variation is due to the range of responses from novice and expert subjects: the subjects with previous experience of formal specifications had the least trouble which can probably be explained by their familiarity with tree representations of terms.

During the exercise, subjects knew they had to define equations in an inductive manner, i.e. by considering the behaviour of the operation for the push and new.stack cases. This is obviously a result of their general computer science training but could also have been reinforced by the pattern in the equations developed in the tutorial.

Several subjects said that they found it difficult to build equations in a top-down manner as advocated in the tutorial. For example, when trying to build the term size?(push(a,s)) these subjects built the push sub-term and attached it to the root node only to discover that they could not insert the size? operation before the push node. Since VISAGE also supports this mode of construction, the subjects who were frustrated by the top-down approach described in the tutorial were told the more general approach and found that it satisfied their previous reservations.

VISAGE provides graphical feedback about whether a plug can be inserted into a particular socket (i.e. whether the sub-term is type-compatible with the particular node in the equation tree). Subjects found this particularly useful, rating it highly in the questionnaire (μ=83%, σ=17%). The feedback helped to resolve problems when subjects tried to add a term only to find that socket would not accept it: *"Of course it doesn't fit! It needs a stack"*.

Look and Feel of VISAGE

The post-exercise questionnaire contains several questions about the general look and feel of the VISAGE system. Although these are generally aesthetic concerns, they are nonetheless important when considering the user interface. In response to a question on the general layout of the display, all subjects found it to be reasonably good, (μ=75%, σ=8%). However, certain aspects could be improved and they are discussed separately below.

One of the main design decisions in building VISAGE is to allow several aspects of the ADT to be shown concurrently, using synchronised views. Consequently, it is imperative that the user does not feel lost when presented by these views. Having each view labelled with its name helped remind the user of the view's purpose and contents. The average rating to the question on how quickly the subjects understood the use of each view was high, (μ=73%, σ=22%). As might have been expected, the high variation is this response was due to novice subjects needing longer to understand the use of each view than the expert subjects. One subject commented that having a logical progression of views of more detailed levels of abstraction and having laid them out in a chain helped in following the development of a specification. As he stated in the post-exercise questionnaire, the best aspect of VISAGE was its *"... adherence to an integrated philosophy"*.

VISAGE is implemented in Smalltalk-80 and so it is important in evaluating the system to separate performance deficiencies due to the interpretive nature of the language from any

deficiency in VISAGE itself. The questionnaire response showed that subjects did not feel the speed of the system interfered to a noticeable extent with the task they were performing, (μ=77%, σ=20%). The variation in this response correlates well with subjects' experience of computer workstations: experienced users were more aware of shortcomings in computer speed.

Since VISAGE is a system that is *driven* by user commands, it is important that the user feels in control of the system instead of the other way round. The questionnaire has two questions to check this aspect of the interface. The subjects felt in reasonable control of what the system was doing (μ=77%, σ=12%), and once they had issued a command, the change in the display was understandable and consistent with their expectations (μ=81%, σ=13%). The only aspect of the system that gave the subjects the feeling that they were not in control was the mechanism for re-drawing the displays according to the view's notion of what a good layout should be. Several subjects spent considerable time arranging the position of icons and did not like the system rearranging this layout whenever the specification was updated. The feedback on this aspect resulted in the mechanism being changed in the updated version of VISAGE to keep a user-defined layout (during a particular session) and to only produce a system-generated one when explicitly requested.

Use of the PLAYPEN

All subjects found the PLAYPEN to be useful in understanding how an ADT behaved, giving it high ratings in the post-exercise questionnaire (μ=88%, σ=11%). Several subjects even thought it was the best aspect of VISAGE, despite it being an early prototype.

The main use of the PLAYPEN for the subjects (as encouraged by the tutorial) was to try out their specifications to see what they did, or to confirm their intuitions about an ADT's behaviour. After the tutorial introduction to building equations graphically, subjects had no trouble building terms since the same mechanism is used. The continuously-updated textual representation of the term being built allowed mistakes to be quickly rectified and gave the subjects confidence that they were building the correct term. This confidence was demonstrated by their use of the more powerful graphical editing commands (such as COPY SUBTREE), that were not covered in the tutorial, to construct terms.

Several subjects correctly guessed (and then exploited) that the text views above each graphical view could be used to edit a term textually as well as simply giving a textual representation of the term. However, because it is easy to make syntax errors (mismatched parentheses, for example), these subjects graphically constructed a skeleton term and then made textual simple changes to it. This is a perfect example of the power of multiple, synchronised representations being used where they are most appropriate. The graphical view is best at defining the skeletal structure of a term without troubling the user

with concrete syntax. This is because graphical editing commands that work on tree structures are provided (such as deleting, moving and copying sub-trees) allowing the user to manipulate the structure at a higher level. Once the skeleton is formed (with the leaves of the term being left unspecified), textual editing can make small, local changes that are more cumbersome when done graphically, such as replacing unspecified nodes by variables.

The evaluation exercise asked the subject to guess the function of an operation that was only described by its equations. As expected, subjects not experienced in recursive definitions found the textual form difficult to understand, although some did guess (but were not sure) that the operation concatenated stacks. The subjects with experience of such definitions correctly guessed the functionality from the text. All subjects, however, used the PLAYPEN to watch what happened to concrete example data that they constructed. As an expert subject commented: *"It was fun to display visually what I am already familiar with textually"*. For an inexperienced subject, using the PLAYPEN was more enlightening: *"Ah! So that is how the recursion works!"*. After using the PLAYPEN, all subjects were confident they knew what the operation did and successfully renamed it to something more descriptive.

The PLAYPEN allows a term to be rewritten according to the equations of the current specification. All subjects used the two commands that implement this facility (one does a single reduction whilst the other repeatedly reduces the term until the canonical form is reached). The two commands were used in different ways. The single-step command (REWRITE) was used several times in succession to show each step of the reduction when the behaviour of the operation was still unknown. The highlighting of the sub-term being reduced was found to be useful in seeing how the selected equation worked. The repeating version (REWRITE ALL) was used to get the final answer when performing quick checks, e.g. to confirm the behaviour of an operation guessed after using REWRITE. As one subject put it: *"It is great to be able to execute specifications. The PLAYPEN is useful because it checks that I have covered all the cases"*. This is interesting because the PLAYPEN does not actually perform this check: the comment does however suggest that the subject is confident in using the PLAYPEN to verify their specification.

Although the PLAYPEN was found to be very useful, it was the least developed part of the VISAGE system and several subjects suggested possible improvements and additional features for a later version. A subject with previous formal methods experience thought it would be useful to be able to *select* the particular sub-term to be reduced and even to choose between several evaluation strategies! The main drawback according to several subjects in using the PLAYPEN was having terms overwritten when single-stepping through a sequence of reductions: if a term T_1 is rewritten to T_2 which in turn is rewritten to T_3, the original T_1 term is overwritten since the view now shows T_3. If term T_1 was

complex, the effort required to rebuild it is considerable. It is also no longer possible to compare an intermediate term with the original, say T_3 with T_1, which reduces the benefits of PLAYPEN. This problem could be minimised if the terms were copied into graphical scrapbooks or if the PLAYPEN maintained a history of the rewritten terms allowing the option of examining any of the intermediate terms.

Error Handling

Although VISAGE reports error conditions if they arise during execution, they are a rare occurrence. This is a consequence of designing VISAGE to prevent the user from getting into error situations wherever possible, instead of merely trapping them once they have occurred. Shneiderman (1983) regards the rarity of error situations as a general characteristic of Direct Manipulation interfaces. In the evaluation only one subject managed to produce an error (attempting to give an operation the same name as its parent ADT). The subject was unaware that this was an error and followed the recovery mechanism without any assistance.† The handling of errors conditions is not covered in the VISAGE tutorial but is similar to the one used in the Macintosh computer which is familiar to the particular subject involved.

Display of VISAGE State Information

The post-exercise questionnaire tried to determine the usefulness of the mechanisms for displaying the current state of VISAGE, in particular, the different cursor shapes and the thermometer-like progress indicator. In general, the response shows that they are useful (μ=71%, σ=20%), but several subjects made specific suggestions for improving their usefulness. Although the cursor was generally the focus of the subject's attention, the different cursor shapes were too small to be noticeable and could easily merge with the background, requiring wild mouse movements before the subject re-located the cursor on the screen. The progress indicator was useful in providing reassurance that something was happening but was also too small and not at the subject's focus of attention.

Exercise Results

All the subjects completed the evaluation exercise, but with varying degrees of ease which is reflected in the variation in times taken to complete the exercise (μ=31 minutes, σ=8 minutes). Most subjects produced a correct solution, with the others dropping marks for relatively minor errors such as failing to give a comment to an operation: the marks out of 10 were μ=9.6, σ=0.6.

---

† The error handling mechanism is described in chapter four.

Overall Reaction

The final four questions in the post-exercise questionnaire were taken from Shneiderman (1986, p.401), and were used to solicit an overall reaction to using VISAGE. The subjects had to give responses to questions with the following anchor points: *Terrible* to *Wonderful*, *Frustrating* to *Satisfying*, *Dull* to *Stimulating* and *Difficult* to *Easy*. The responses to these questions are of little use except in indicating the general feeling of the subject to using the system. With little variation all subjects gave the system high ratings to all questions (μ=80%, σ=13%). It can therefore be safely concluded that the subjects liked using VISAGE.

Results of the Control Group

The only subject from the control group to complete the task had prior experience of formal specification but did need two attempts at deciding what the MYSTERY operation did before feeling happy with his answer. It was interesting to note that this subject used examples, expressed using a personalised graphical notation, in reaching his answer.

Although the three novice subjects all managed to define the new operation (albeit with syntactic slips), the equations gave more problems. One novice got one equation correct but gave up on the second. The other two novices became confused and eventually gave up in frustration. None of the novices felt able to say what the MYSTERY operation did. These results are reflected in the responses to the questionnaires on their confidence: three subjects said that their confidence of writing and reading algebraic specifications had not changed; one novice said that his confidence of writing such specifications had actually decreased.

7.5 Limitations of the Evaluation

It is not good enough simply to let users play with the system and then ask them what they think. To be of any real use, a representative group of users should be subjects in a controlled experiment designed to investigate particular aspects of the system. An ideal evaluation of VISAGE would involve a comparison with other mechanical and manual methods of introducing the formal specification of ADTs. This ideal is difficult to achieve because of pragmatic considerations. For example, it would be necessary to have a *large* pool of subjects of equal experience and backgrounds that could be partitioned equally into subject and control groups. A more comprehensive evaluation task would need to be devised that was not biased towards one group or the other. As Sheil (1981, p.103) regards this as a difficult task: *"The construction of realistic programming situations that differ in only (a few) controlled ways is ... both difficult and subject to its own forms of bias"*. This problem is compounded because a full evaluation of VISAGE would require the

involvement of specialists in fields such human-factors and graphic design, for example. Such an evaluation is beyond the scope of this work.

As a practical attempt at evaluating VISAGE, ten subjects were involved (not counting those involved in the pilot study) in the evaluation with four in the control group. These groups are obviously not large enough to obtain quantitative results with a high degree of statistical significance. However, they are large enough to identify *trends* in subjects' performance and reaction to using VISAGE, identify the most serious problems with the system, and to provide *qualitative* results of subjects' preferences. For this reason, the numerical data for averages and deviations in subjects' responses in the *Results* section above are given only to show the trend and not to support more general statements about VISAGE. However, the marked differences between the evaluation and control groups does suggest that VISAGE is an improvement over conventional methods of introducing novices to the formal specification of ADTs.

It is important when conducting an evaluation that the subjects be representative of the intended user population. It is expected that VISAGE users would be inexperienced in algebraic specification but knowledgeable about general computer topics such as basic data types and the advantages of modular design. Unfortunately, a large group of such users could not be found: the subjects involved in the evaluation of VISAGE had a wider range of experience of algebraic specifications. Although this has the advantage of getting the responses of different types of user, it also means that the accuracy of the responses for novice users will be reduced.

VISAGE can be regarded as a graphical tool that assists a user in developing their own intuitions about how to specify the interface and behaviour of an ADT in a formal way. However, in a contemporary environment, such graphical tools will not be commonly available (at least in the short-term), and so specifications will still appear in a purely textual format. It is hoped that after extensive use of VISAGE, the user will have acquired enough skill and confidence to specify and understand an ADT using text alone. Unfortunately, this process will take more time than that allocated in the evaluation, and so it is premature to test the subjects for the acquisition of such skills.

The exercise used in the evaluation of VISAGE was kept short so that subjects could complete it in a reasonable amount of time. However, this means that any benefits or problems with the system when used on much larger specifications would remain undetected. To overcome this limitation, VISAGE is to be used by a subject with previous experience in formally specifying software to develop larger and more complex specifications of ADTs than the ones possible in this evaluation. This obviously moves away from using VISAGE as a tool for introducing specifications to novices and more towards using it as a development tool.

The tutorial introduces the subject to only a subset of the VISAGE commands to keep the evaluation practical; however, these commands are sufficient to allow reasonably complex specifications to be constructed and manipulated. Although many commands were not introduced, confident subjects often experimented with them after seeing them on the menus. However, several commands and facilities remained unused and so remain outside this evaluation. More advanced use of VISAGE in handling more complex specifications, as described above, will go some way to overcoming this limitation. The untested facilities are:

- construction of equations with conditional right-hand sides;
- parameterised specifications and the enrichment of existing specifications;
- use of the Dependency View to explore relationships between operations;
- support facilities such as icon editing, screen dumping of graphical views, pretty-printing and type-checking of specifications.

7.6 Summary

This chapter has described the method and results of conducting an evaluation into the benefits of the VISAGE system for introducing formally specified ADTs. Despite the limitations in the evaluation, it helped to identify the weaknesses and strengths of the system that will influence the design of the next version. The most basic result of the entire evaluation was how it confirmed Brooks' statement, given at the beginning of the chapter, that the results of an evaluation will surprise the system's designer. The evaluation of VISAGE showed that no user interface design can please all of the users all of the time. A successful system (i.e. acceptable to all or even most of the users) will be based on design compromises and will only be reached by a process of repeated refinement and evaluation. Unfortunately, it is all too common for computer systems to be imposed on users without it having any evaluation let alone an iterated one.

Subjects from a wide technical background were able to learn how to use VISAGE to create and manipulate formal specifications of a simple but useful ADT, in a short period of time. This includes several subjects with absolutely no previous experience of formal specifications. A typical statement was: *"I've never been happy with ADTs before but VISAGE was fun to use: I enjoyed it"*. Two subjects with experience in this area (one being a lecturer) thought that VISAGE would be a useful tool for teaching novices the basics of algebraic specifications: *"Maybe this type of program will encourage other, less mathematically trained, people to use precise descriptions [of software]"*. This is confirmed by a subject with a traditional programming background: *"I would like tools like VISAGE to learn formal methods as I know they are the way things are going but I am put off by all the complexity"*.

Chapter Eight
Conclusions and Future Work

The main aim of this thesis, as stated in chapter one, was to investigate whether the formal specification of ADTs would be made more palatable to novice users if they were incorporated within an interactive environment that used graphical representations as wells as textual ones. This chapter presents the conclusions of testing this thesis.

8.1 General Comments

As far as is known, VISAGE is the only system that supports all aspects of the algebraic specification of ADTs within a graphical framework. Consequently, it is impossible to draw conclusions about the success of VISAGE by comparing it with related systems. Instead, we must be content with looking at VISAGE in isolation, using criteria such as how well it supports the various aspects of specification and whether or not it is accepted by its intended users.

When the system was first being considered there was uncertainty as to whether it would be practicable to build such a system. VISAGE may therefore be regarded as an existence proof that the marriage of computer graphics techniques and formal specification is indeed fruitful. Furthermore, the evaluation of VISAGE provides empirical evidence that the approach was preferred by users to the conventional, text-based method of introducing formal specification. Unfortunately, it is difficult to measure objectively any improvements in user performance that might have resulted from using VISAGE. However, there is strong anecdotal evidence that insight and confidence are enhanced through the use of VISAGE, but it will require experiments on a larger-scale in order to measure objectively any improvement in user performance.

As VISAGE evolved, it became clear just how important examples were in providing insight into how an ADT behaves. This is reflected in the increasing emphasis within VISAGE on allowing users to experiment with terms they had created themselves using the visual programming facility and the PLAYPEN. Although it is beneficial to use visualisation techniques to view data in a new way, to maximise the pedagogical benefit it is necessary to provide some way of interacting with the data through the graphical view: understanding comes from writing as well as reading. The VISAGE experience of specification *with* examples seems to support Piaget's theory† that people naturally learn

† Piaget's theory is discussed in Papert (1980).

by starting with concrete examples and working towards the abstract. Teaching systems should therefore make the manipulation of concrete examples their basic activity.

8.2 Visualisation Problems: Revisited

Chapter three identified several problems that plague the majority of visualisation systems: graphical representations are less compact than their textual equivalents; have poor run-time performance; and often use obscure graphical representations. Does VISAGE suffer from any of these problems? Each of these problems will be re-examined from the perspective of the VISAGE system.

8.2.1 Too Much in Too Little Space

Many visualisation systems suffer from cluttered displays because they combine information about the internal structure and external connections of an object in a single representation, i.e. they fail to exploit the inherent abstractions. VISAGE avoids the problem of an individual display becoming cluttered by having multiple views, each displaying a single facet of the underlying object. In this way, the total information load is considerably reduced.

Although VISAGE benefits from its use of multiple views, it also takes advantage of the modularity of the ADTs and their specifications. ADT specifications are usually smaller than the equivalent programs and so their representation will be correspondingly more compact.

8.2.2 Poor Interactive Performance

A common reason why visualisation programs are slow is because they attempt to compute the optimal layout for a particular structure according to some fixed aesthetic criteria encoded within the program. If they did produce the optimal layout then the wait would probably be worth it. Unfortunately, aesthetic judgement is a personal, subjective matter that is impossible to encode in a program. Moreover, the algorithms developed for generating good layouts are complex with poor run-time performance making them unattractive in an interactive environment. The conclusion reached in developing VISAGE is that the user knows best. No matter how good the layout algorithm, the user will want to tweak the output according to personal taste. Given the users' desire to customise the display, the most practical solution is to get a reasonable representation on the screen within a consistently short time and let the user modify it until they are satisfied with the result. The system should then present this layout whenever the user selects it in the future.

This basic layout mechanism could be improved in a couple of ways. Firstly, the system could infer certain layout rules from a user-modified display and use them when creating

similar displays. This approach was used with limited success in the PERIDOT system, Myers (1987). Alternatively, the user could specify how much effort the layout algorithm should invest in producing a display, for example, either quick-and-dirty output for use in an interactive setting, or high-quality rendering that could be used as documentation.

Chapter three identified the complementary problem of visualisation systems being unsuccessful because of the inefficient mechanisms they provided for handling user input. Although the design of good input techniques is often a matter of common sense, experience with VISAGE showed that subtle aspects of the interface can have major effects on how users feel about the system. These problems were certainly not anticipated in the original design and only became manifest during trials of the system with real users. These observations reinforce the need for the *iterative* design of user interfaces using the positive feedback from the evaluation studies.

8.2.3 Incomprehensible Graphical Representations

Despite the impassioned claims of their proponents, visualisation systems that provide a single view of program or data have often failed to be accepted because although they were good at showing one aspect they were poor at other, equally important aspects. Users generally found this blinkered view too inflexible for their needs and therefore stuck to their inefficient but general textual methods. Although VISAGE provides graphical representations for all aspects of specifying ADTs it does allow the use of conventional textual methods if the user feels they are better suited for a particular task. Users should be able to switch between text and graphics as confidence and circumstances dictate without explicitly switching modes. This approach was very successful with even ardent advocates of text finding that a combination of text and graphics was a more effective method of interaction.

8.3 Classifying VISAGE

Myers (1989) claims that any system that produces a graphical representation of an essentially textual program is a program visualisation (PV) system. VISAGE clearly falls into this category as it provides graphical representations of text-based algebraic specifications. Such specifications fit within his definition of the term "program" being a formal description that can be executed by a machine. Myers further partitions these systems according to whether they illustrate the code or data of the program. Such a distinction is irrelevant in the case of algebraic specifications of ADTs (as well as languages such as LISP) since program code and data have the same basic form. His further refinement according to whether the display is static or dynamic is also irrelevant in the case of VISAGE since it provides both kinds of display, in the PLAYPEN, for instance.

Myers (1989) regards visual programming (VP) systems, which allow a program to be specified in two or more dimensions, as being completely different from PV systems. The basis of the distinction is Myers' assertion that *"If a program created using VP is to be displayed or debugged, clearly this should be done in a graphical manner, but this would not be considered PV"*. This rather subtle distinction falls apart in the case of VISAGE which satisfies the criteria for both PV and VP. The reason for the problem is the way VISAGE supports *both* textual and graphical representations of a specification: it is a PV system because of the way it can illustrate a textual specification; it is a VP system because it can create such a specification graphically.

The problems of trying to classify VISAGE according to the taxonomy in Myers (1989) raises questions about what it means to use graphics in programming. Clearly, the meaning of the term *program visualisation* as defined by Myers is too weak to handle systems, such as VISAGE, that support multiple representations of a program. Its demand that the underlying program be represented textually is artificial and needs to be generalised if it is to remain useful. PV should simply mean a mechanism where graphical representations are used to illustrate some facet of a program, regardless of whether that program's underlying representation is textual or graphical, a mixture of the two, or even some other medium.

8.4 Experience of Using Smalltalk

In chapter four, Smalltalk-80 was selected as the implementation language for the VISAGE prototype. With hind-sight, was this the correct decision? Most emphatically, yes! After being brought up on a diet of barely interactive editors, 24-row alphanumeric terminals and the delays of iterating around the edit–compile loop, the Smalltalk environment, with its highly interactive, graphical user interface, is a programmer's Utopia. Smalltalk's programming support tools made program development an efficient and enjoyable process. The large collection of pre-defined classes (especially those supporting interactive, graphics-based applications) allowed prototype programs to be written quickly, encouraged by the Smalltalk philosophy that "plagiarism" is productive. Being able to build programs quickly meant that many more alternative ideas could be implemented and tested than would have been possible in a more conventional and rigid program development environment such as UNIX or worse.

However, there were a number of problems in using Smalltalk. Several of these were anticipated in the discussion in chapter four. The main difficulty in the early phases of learning Smalltalk was the need to reconsider the meaning of previously understood concepts such as "program", "operating system" and "environment". In the conventional imperative programming style, programs are self-contained units with clearly delineated boundaries between the program, the supporting subroutine libraries, and the operating

system. In Smalltalk this distinction has been removed, resulting in a single homogeneous architecture with the result that user programs are indistinguishable from "system" ones. It becomes impossible to isolate a program or even to know "where a program starts". Although elegant and powerful, this freedom is intimidating after the oppressive nature of conventional systems.

A further problem was due to the lack of guide-lines on how to build programs using Smalltalk. Much experimenting and reading of system code† was needed before programs were being written in the "approved style". Fortunately, the Smalltalk language itself was very easy to learn thanks to its simple, consistent syntax and relatively clean semantics.

One part of the Smalltalk system that was found both useful and arduous was the Model-View-Controller (MVC) framework for constructing interactive applications. Although the MVC lies at the heart of the Smalltalk environment and is the "official" framework for building interactive applications, there is a dearth of tutorial material on how it should be used. Smalltalk has a reasonably large collection of views and associated controllers that provide a starting point for building user interfaces (UIs). Unfortunately, this collection is obviously directed towards supporting the text-oriented tools of the Smalltalk environment rather than supplying more generic UI components. For example, common components such as sliders, dials, versatile menus, and two-dimensional scrolling facilities are sadly absent and had to be built from scratch during the development of VISAGE. However, it is one of Smalltalk's strengths that it was possible to construct the new components at all.

Smalltalk's UI classes appear ad hoc when compared to the comprehensive set of Collection classes which provide elegant and powerful data structuring facilities. In their defence, the Smalltalk designers freely admit‡ that the Smalltalk UI mechanism was an experiment that was only partially successful. Even so, although the Smalltalk MVC framework has its problems it is easier to use, despite being decade-old technology, than the rather baroque contortions needed to use contemporary UI libraries such as SunView for Sun workstations.

---

† At the time this was perhaps the only way of learning Smalltalk given the lack of tutorial material. Luckily, many texts have recently been published that introduce programming in Smalltalk. In addition, the latest version of Smalltalk-80 now comes with an on-line tutorial that includes embedded code fragments for the user to execute.

‡ Most commonly within the source code comments of the system itself. Perhaps the most telling admission, however, was the non-appearance of the much-publicised book in Addison-Wesley's Smalltalk series on designing interactive applications using the MVC.

8.4.1 The Need for a Linkage Component

In an undisciplined framework such as Smalltalk's MVC, the developer is given very little support in deciding how to partition the functionality of his application among the components of the framework. For example, how much should the Model know about presentational aspects and vice versa. Not having strong guide-lines about where certain components belong often leads to unstructured, monolithic programs that are difficult to maintain. One general solution to this problem is to introduce a *linkage component* (LC) that separates and is knowledgeable about both the model and its user interface while they remain ignorant of each other (Cockton, 1987). In this new architecture, the LC alone has access to the model, moderating all data accesses to it. The introduction of this new component results in the modified framework illustrated in figure 8.1 (compare it with the original in figure 4.1). Although VISAGE started out adhering to the prescribed MVC framework, a linkage component naturally emerged in order to overcome certain problems encountered during the implementation.

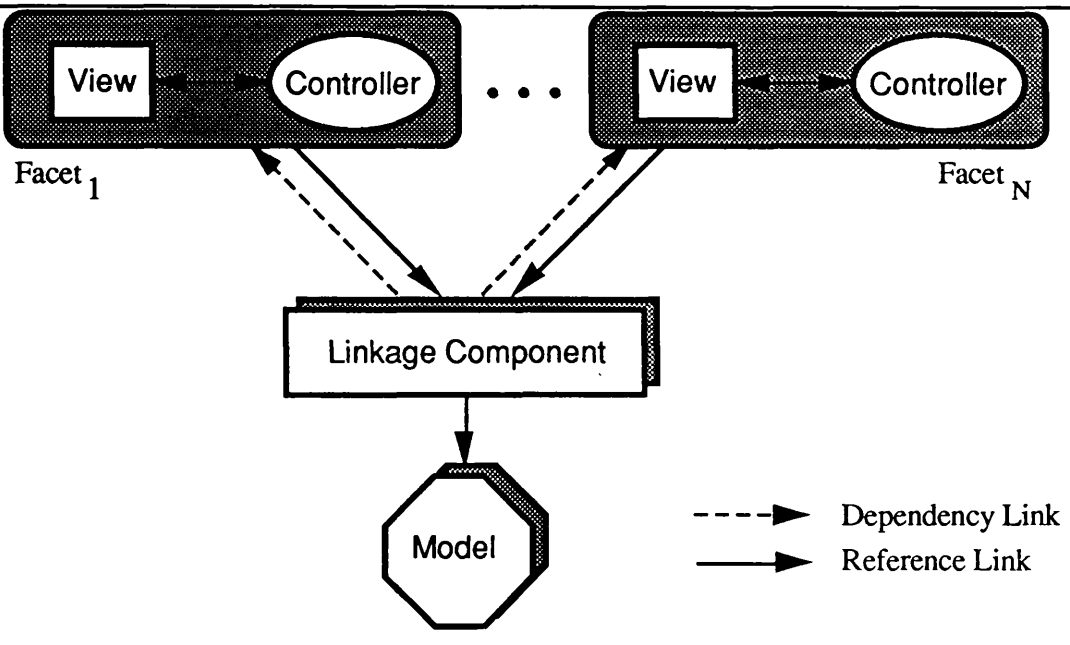


Figure 8.1: The MVC Framework with Linkage Component

The main consequence of this new separation is that the ADTs need not know they are being visualised and so their implementation need not provide any support for it. As well as acting as a data switch, the LC performs a large amount of the ancillary computation required to support the views, e.g. performing consistency checking and handling semantic feedback.

The linkage component provides the added benefit of acting as a convenient repository for data that must be shared between the various views that it supports. For example, if an

entity, such as an operation, is to be represented in several views it is obviously desirable that the same icon is used for each instance, particularly if that icon's image has been created by the user. The icon must therefore be shared between all the views that use it, rather than being local to one particular view. The prescribed way of achieving this in Smalltalk† is clumsy being extraneous to the normal accessing mechanism. However, the LC, being equally and easily accessible to all the views, is a natural choice for the placing of this shared resource.

8.5 Further Work

The VISAGE prototype is very much an exploratory investigation into visualising and visually programming ADTs. The results of using VISAGE identified certain areas that would benefit from deeper investigation. Although some suggestions for future development have been given in previous chapters, further suggestions are discussed below.

8.5.1 Alternative Graphical Representations

The VISAGE graphical representations evolved to meet the goal of presenting the necessary information in a perspicuous format within the constraints of efficiency and practicability. However, they are by no means the definitive solution to this problem. Other representations, possibly of radically different character, could present the same information, perhaps with greater clarity thereby offering a deeper insight into the nature of ADTs.

An interesting future experiment would be to develop and then evaluate alternative graphical representations. Two particular avenues of exploration, namely the use of colour and shape, are discussed in greater detail later in this section. However, other attractive approaches include, for example, greater use of animation to illustrate the rewriting of terms, and the use of physical containment to handle the nesting of detail in terms. Once several alternative representations have been built and evaluated, it should be possible to form a general theory of what makes a good graphical representation of algebraically-specified ADTs. Such an experiment should be relatively simple to perform due to the modular architecture of VISAGE (shown in figure 5.1) which allows new views to be added (or substituted for old ones) without affecting any existing views.

It may be suggested that the current set of views in VISAGE do not make use of the many graphical attributes that an object may possess if rendered using very high quality display technology. Such attributes includes texture, transparency, depth and lighting effects. Although rendering these attributes is computationally expensive, the increasing avail-

---

† It involves using *pool dictionaries*: a poorly documented way of providing selective global variables.

ability of powerful graphics workstations now makes them practicable. There is considerable potential for applying them to visualising ADTs. For example, by adopting the ideas of Reid (1989), the size of icons could indicate the complexity of the associated object according to some measure, e.g. number of operations or equations in an ADT's specification. Investigating the use of these additional graphical dimensions seems a promising avenue.

8.5.2 Use by Experts

The current version of VISAGE is designed for use by people who have little or no experience of formally specifying ADTs. The main reason for targeting this group is because they stand to benefit most from a pedagogical tool such as VISAGE. However, a secondary reason is that visual programming systems often cannot support the size of problem tackled by expert users and so must be content with handling the more trivial examples encountered by novices. As has been argued, the nature of ADTs and the design of VISAGE itself, has moderated the effect of these problems in this circumstance. It would therefore be an interesting experiment to apply VISAGE to the larger and more sophisticated specifications tackled by expert users in order to locate any inherent weaknesses.

How would VISAGE need to be changed if it were to support such an experiment? Compared with a novice, expert users typically deal with larger collections of specifications and use more sophisticated language constructs in order to handle a richer variety of problems and associated types. This demands that the simple textual specification language developed to exercise VISAGE will need to be considerably extended to give it the descriptive power of, say, OBJ2 (described by Futatsugi et al., 1985). Examples of features to be included in the new language are improved error handling; letting operations return more that one result; and allowing mixfix syntax† rather than the hard-to-understand but easy-to-parse prefix notation currently used in VISAGE‡.

These language extensions will, in turn, require extensions to be made to the graphical views of VISAGE, in particular the Hierarchy View. For example, because software engineers often create new artifacts by building on top of existing ones, an early extension to VISAGE would allow a new ADT to be specified by enriching an existing one with new operations or behaviour or both. An example of this would be to enrich a list with a

---

† This permits the declaration of operators using prefix (e.g. *sin* x), post-fix (e.g. n*!*), or in-fix (e.g. 1+2) notations. In this general case, the operators are not restricted to simple identifiers but can be any symbol (within the limits of the language).

‡ The VISAGE language does use mixfix notation for the pre-defined conditional operator *if...then...else...*

sorting ability to create an ordered list. Clearly the original and enriched ADTs are related and this should be graphically represented in the Hierarchy View.

The VISAGE user interface has been designed to make it simple and intuitive for novices to create, manipulate and experiment with ADT specifications. To satisfy this requirement, the interface has adopted a supportive, forgiving style. For example, heavy and consistent use is made of pop-up, context-sensitive menus for grouping commands applicable to a particular entity; a user error, when trapped, gives rise to a comprehensive error message describing how to correct the mistake. Typically, expert users do not like such succour as it tends to slow down their progress. They much prefer terse but informative error messages and accelerators for issuing commands. However, comprehensive support should be available if the user (no matter how "expert") needs it.

In general, as users become more expert they tend to prefer text-based interaction styles to the slower graphical ones. For example, in a word processor an expert typist may prefer using keys that move the cursor to the slower equivalent of selecting the position by using the mouse. If a graphics-based application intends to support expert users then it must provide graphical interaction techniques that are more powerful (i.e. more done in a period of time) than text editing techniques. This is the principle behind the commands in the Equation View for manipulating entire sub-trees as a single unit.

8.5.3 Handling Data Type Parameters

Although VISAGE allows the specification of parameterised ADTs such as List[Elem] it does not allow the instantiation of these generic types, e.g. to substitute the actual type Natural for the formal type Elem. Such a mechanism would be very useful for constructing example terms in the PLAYPEN.

A natural extension to the facility for parameterising ADTs is a way of specifying restrictions on what types can be used to instantiate the parameter. For example, consider an ordered list ADT where the type of the members of the list are specified by a parameter called Elem. To allow the list to be ordered it is necessary that the actual type substituted for Elem have an ordering relation over the values of the corresponding sort. It is desirable to make such a constraint explicit within the specification; an example specification header based on the notation used in Cohen et al. (1986) is:

```
Datatype: Ordered-List [Elem with {less?: Elem × Elem → Boolean}] ...
```

The braces following the parameter name (Elem) contains a set of signatures of the operations that any actual type must provide. In this case it specifies that a binary predicate

`less?` must be provided†. When the user tries to instantiate this type (perhaps using a variant of the plug-and-socket metaphor introduced in chapter six) the system will ensure that this constraint is satisfied. This constraint on the parameter could be handled by a simple extension to VISAGE: the Hierarchy View's display of the above header would be similar to that shown in figure 8.2.

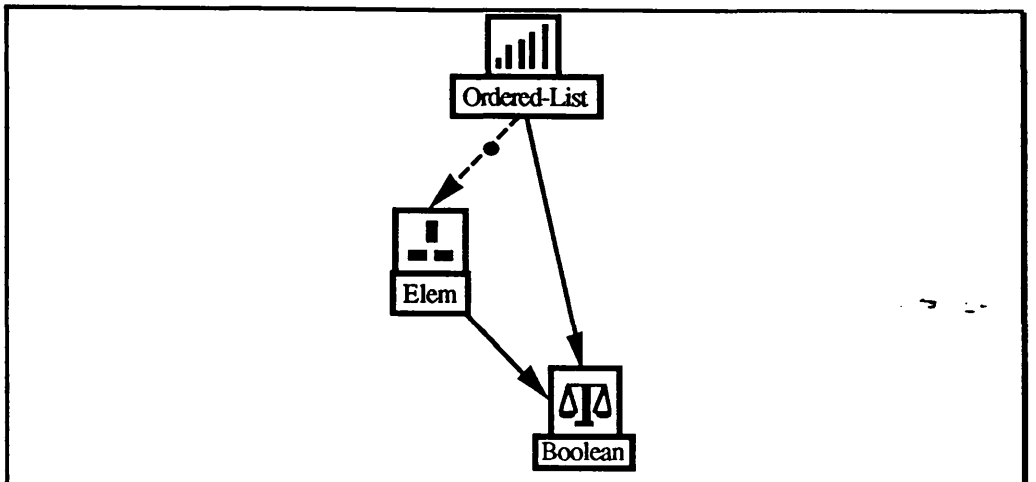


Figure 8.2: Hierarchy View of a Parameterised ADT

The dot on the link between the Ordered-List icon and its parameter's icon indicates that there is a constraint on the actual types that can be substituted for Elem. If Elem were selected for display, it would have the following specification:

```
Datatype parameter:  Elem uses: Boolean;
less?:  Elem × Elem → boolean;
EndSpec
```

The new qualifier *parameter* distinguishes this specification from that of a conventional ADT. This specification gives rise to a conventional Signature View but since there are no constraints on the behaviour of the operation (i.e. no equations), the Equation View remains empty.

8.5.4 Specification by Example

VISAGE allows users to experiment with the specifications of ADTs by creating examples of the values of these types and then apply existing operations to them to confirm that these operations behave as expected: specification-*with*-examples following the terminology of Myers (1989). An interesting extension would turn this sequence on its head to provide a specification-*by*-example facility: the user could define an operation by *showing* the

† Note that the semantics of this operation is not specified.

system what the results would be when if the operation were applied to a selection of example values. A selection of examples will be necessary since the operation will have to be defined for the different cases, e.g. in the stack ADT the operation needs to be applied to `new.stack` and then a `push` term. In general, each case would give rise to a new equation. VISAGE could ensure that the operations were fully defined by checking that the user had supplied examples for each case.

As an example of defining an operation, consider the situation in figure 8.3 where the behaviour of the stack operation `pop` is being defined in this manner. Here the left-hand term is the *before* state with the *after* state shown on the right-hand side.

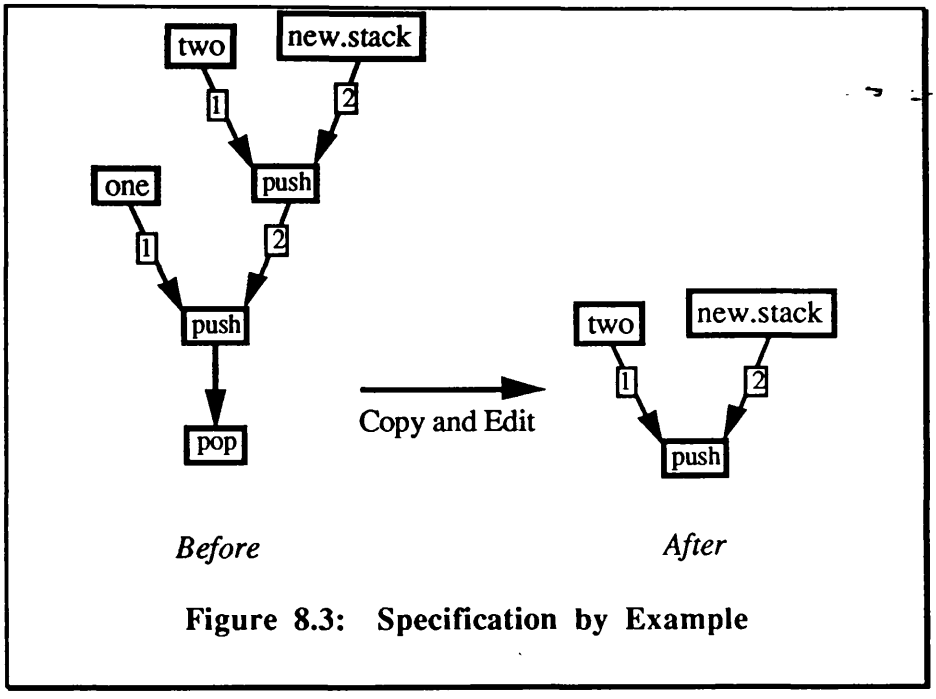


Figure 8.3: Specification by Example

The mechanics for defining the operation are a simple extension of the existing PLAYPEN editing facilities. The user creates a term with the operation being defined at the root. This ensures that the operation has the correct number and type of arguments. The term need not be totally defined so long as the undefined parts are not involved in the operation. The user then places in the PLAYPEN in a new *specify* mode. This causes the PLAYPEN to create a copy of the input term without the original root node (i.e. strips away the operation being defined so as to prevent circular definitions). The user then edits the copy to produce the desired result using the normal editing facilities of the PLAYPEN. To prevent the introduction of free variables (i.e. variables that appear on the right-hand side of an equation but not on the left), the user is prohibited from attaching new variables to the edited term. Once the desired result has been produced the user issues a command to leave *specify* mode. VISAGE then creates a new equation based on the structure of the *before* and *after* terms.

In creating the equation that describes the editing transformation performed by the user, the system needs to use inference rules in order to find the most general rule from the concrete example used. For example, it would be trivial to generate the following equation from the above example:

```
pop(push(one, push(two, new.stack))) = push(two, new.stack)
```

Unfortunately, it is unlikely that this is what the user intended. By a more careful analysis of the *before* and *after* terms the system could infer the desired equation:

```
pop(push(a, s)) = s
```

However, before making this assumption, the system could request the user to supply one or more additional examples in order to check that the general equation does indeed cover all the examples. For each example, the system would show what the result would be if the suggested equation were applied. The user would then either accept the rule or correct the output. If the output needed correction then the system would modify the equation to include this new case. Goguen (1986) suggests a way of avoiding the need for the system to infer a general rule from concrete examples. By making the user manipulate *generic examples*, the system can generate an equation directly since whole families of terms are being represented by a single term. Although such an approach may be desirable for expert users, experiences of using examples in VISAGE indicates that novices would have difficulty in generalising their examples and would much prefer to supply several concrete examples rather than one generic one.

Halbert (1984) states that as well as difficulties in inferring a general rule from several concrete examples, programming-by-example has difficulties in handling branching and iteration. Although application of the technique to algebraic specification does not involve iteration it can involve conditional terms and the basic specification mechanism above would need to be extended handle this. Although Halbert suggests one approach, the problem is by no means solved.

8.5.5 Adding Colour

The sensible use of colour in user interfaces offers the system designer a new dimension with which to represent an object's attributes and the relationships between objects. This dimension could not be explored in the VISAGE prototype since only monochrome work-stations were available at the time. In addition, the standard Smalltalk system does not support colour within its display model. These restrictions meant that occasionally a somewhat clumsy graphical representation had to be used for a particular attribute when colour encoding would have been more elegant. For example, an early version of the Dependency View tried to distinguish between observer, constructor and mutator operations by assigning different border patterns to their icons. Since the icons were quite

small, these adornments were often over-looked. A better solution would be to colour the icon according to the category of its operation, e.g. constructors in red, observers in green, mutators in blue.

As a further example, type information is currently conveyed by the (user-defined) icon image associated with each ADT. When building terms, the sort of empty sockets can be determined from their image as it is the same as their sort's ADT, e.g. a Boolean socket would have the same image as the Boolean ADT. However, internal nodes are shown as operation icons with no explicit representation of their sort. If operations were assigned a colour determined by their resultant sort then type information would be visually obvious. Questions of type compatibility when building terms would be transformed into questions of colour identity. When using colour to convey information, it should be remembered that many people (roughly 8% of males) are colour blind and would find general colour matching to be very difficult.

Colour could also be used to represent attributes that are not currently shown explicitly. For example, to discover in which ADT an operation is defined, the user must use the help facility. If an ADT is assigned a particular colour (either automatically or by the user) then all operation icons defined in that ADT could be displayed with that colour, thereby making identification trivial.

8.5.6 Use of Shape

For implementational simplicity, the icons used in the current VISAGE prototype are limited to rectangular shapes. A future development would be to exploit a richer vocabulary of geometric shapes to denote objects of different types. An ADT could therefore be recognised by its shape as well as its iconic image: questions of type compatibility translate into questions of shape compatibility.

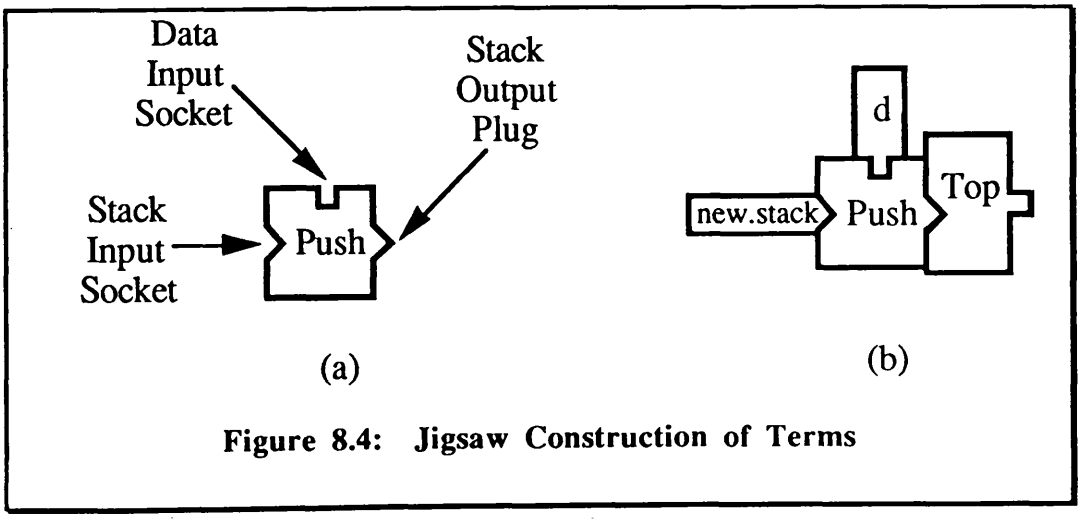


Figure 8.4: Jigsaw Construction of Terms

This idea is an extension of the "plug and socket" metaphor introduced in chapter six. An operation icon has a number of *reaction sites:* each argument to the operation has its own

input site (socket) and there is a single output site (plug) for the operation's result. Each site can only be connected (by juxtaposition) with one of complementary shape. This ensures that terms are always type correct simply by the geometrical constraints imposed during their construction. An example of this physical construction of terms is shown in figure 8.4.

Figure 8.4(a) shows the anatomy of a shaped icon. The concave sites are input arguments for the `push` operation; the single convex site is the output result. Figure 8.4(b) shows the construction of the term `top(push(d,new.stack))`.

This jigsaw metaphor can be extended to represent the concept of sub-typing e.g. the non-zero integers are a subtype of the integers. A value from a subtype can be used whenever a value from the parent type is expected, but not vice versa. This can be represented graphically by the shape for the subtype being similar but not quite the same as the parent's shape: the subtype shape will fit wherever the parent's will fit but not vice versa.

Although this use of shape appears elegant in this example, problems occur in arranging input sites with icons for operations with large numbers of arguments. It is also sometimes impossible to construct a complex term so that it lies flat on a two-dimensional surface due to sub-terms overlapping each other. The basic problem however is in handling the generation, manipulation and interaction of the complex shapes that arise in realistic applications. A similar conclusion was reached independently by Glinert (1987) who used jigsaw shapes to enforce syntactic correctness by having them represent the constructs used in Pascal programs. Unfortunately, this use had only limited success because of the complex set of shapes required to handle the idiosyncrasies of Pascal. The use of shape to convey type information in VISAGE offers greater hope since, in general, only a small number of shapes will be required making both generation and recognition considerably easier.

8.5.7 Integrating VISAGE with Other Tools

The current version of VISAGE is a stand-alone application for introducing the specification of ADTs rather than as a software development tool. However, there is nothing fundamental that prevents VISAGE from being used in such a capacity, although it would require additional implementation effort.

The final output from VISAGE is a collection of textual ADT specifications. If VISAGE were integrated with other automatic or semi-automatic tools, then this collection of ADTs could be passed to them for further processing. For example, automatic translators such as the one described in Zhong *et al.* (1988) could convert these ADT specifications into executable code. Alternatively, the specifications could undergo more extensive testing than is possible with an interactive tool such as VISAGE.

8.5.8 Syntax-Directed Text Editing

The VISAGE Specification View differs from the graphical views in that it does not prevent the user from making syntax and type errors as they type the specification. Although these errors are trapped when the user tries to accept the edited specification, it would perhaps be better if the user were prevented from making them in the first place. This could be achieved, for example, by using a syntax-directed editor. Such editors are relatively common, particularly in teaching applications, and it would be an interesting experiment to integrate one within the VISAGE system. Although such editors prevent the user from making syntax errors they are often inflexible and dogmatic in their adherence to the grammar. For these reasons, syntax-directed editors are usually disliked by experienced users. The system should therefore allow the user to revert to free-form editing once they become used to the language and start feeling overly constrained by a syntax-directed editor.

8.6 Summary of the Thesis

This thesis argues that the application of computer graphics techniques can make the algebraic specification of abstract data types more palatable to novice users. This has been demonstrated by means of a prototype implementation and its evaluation. As far as is known, this prototype is the first example of a system that handles both the syntactic and semantic aspects of ADTs within a graphical framework. To serve this rôle, several specialised graphical representations, both static and dynamic, were developed.

This thesis presents a first attempt at integrating the algebraic specification of ADTs within an interactive, graphical environment. Before considering the union of these two domains, it was necessary to introduce each individually. Chapter two introduced ADTs by considering the benefits they bring to the development of reliable, flexible software. Algebraic specification was then identified as the most common technique for precisely describing and verifying the structure and behaviour of ADTs without regard for implementational issues. However, the chapter recognised that the use of abstract mathematics to specify ADTs would make the technique unattractive to a large number of software practitioners. If the formal specification of these crucial components is to become standard practice, it is necessary to overcome this aversion.

Chapter three proposed that graphical representations offer the chance to describe complex objects, particularly those encountered in software, in a new and insightful manner. Complementary graphical interaction techniques have been developed to create the underlying object through the manipulation of these graphical representations. A survey of visualisation systems of programming revealed that although considerable attention was

given to the latter stages of software development, relatively little had been paid to earlier phases, in particular the formal specification of ADTs.

Having introduced ADTs, recognised the potential of visualisation techniques, and identified the curious omission of applying these techniques to the specification phase of software development, chapter four suggested that the integration of algebraic specification within a graphical framework would be fruitful. Given the absence of a precedent, it was necessary to design a prototype ADT visualisation system from scratch. The principal issues involved were presented in this chapter.

A more detailed description of the design was presented in chapter five. This concentrated on the various graphical and textual representations that evolved to provide a complete description of an ADT. A description of the layout algorithms and major implementation details were presented. This chapter also described the design of a novel integrated help and commenting facility.

Chapter six described in detail how the graphical representations developed in chapter five were used as the basis of a graphical editing facility that allows the creation and manipulation of ADT specifications within a unified environment. The chapter concluded by describing a graphical interface to a term-rewriting engine which allows the creation and transformation of user-supplied example data.

Chapter seven described an evaluation of the prototype ADT visualisation system whose design was described in the previous three chapters. This included a discussion of the experimental design, a presentation of the results obtained, and a review of the limitations of the evaluation. An analysis of the results showed a distinctly positive reaction from the VISAGE group that was completely absent in the control group which supports the stated thesis that the use of computer graphics techniques makes the algebraic specification of abstract data types more accessible and palatable to novice users.

8.7 Conclusions

The algebraic specification of abstract data types is an important weapon in the battle for the construction of correct software. However, the technique will only gain wide-spread acceptance if it is presented in such a way that software practitioners can appreciate the benefits it can bring to their own work. This is unlikely to happen while the technique is dependent on obscure mathematical symbolism. This thesis has shown that an interactive, graphical environment offers one way of making the technique less intimidating and accessible even to users with few formal mathematical skills.

The juxtaposition of textual and graphical representations of an ADT has been successful in allowing novice users to learn about algebraic specification through the manipulation of graphical displays. The use of multiple, synchronised views of an ADT helped users

understand the various aspects of a specification and the relationships between them. These abstractions later simplified the incorporation of a programming facility within the graphical framework by allowing the generic editing mechanisms to be tailored to the facet displayed within a particular view.

Although the tree structure used to represent terms and equations within VISAGE was found by the subjects of the evaluation to be more palatable that their textual equivalents, the representation suffers from its need to deal with the general case. VISAGE therefore complements those systems (such as BALSA (Brown and Sedgewick, 1984)) where hand-crafted, perspicuous representations are used to give insight into the behaviour or form of particular cases.

The evolution of VISAGE has been marked by an increasing awareness of the importance of examples in helping to understand the behaviour of an ADT. An experimenting-with-examples facility appears to be essential in a system intended to introduce a concept, especially if that concept is of an abstract nature. Graphical interfaces are particularly good for the manipulation of examples since they minimise syntactic overheads, allow the structure of examples to be easily discerned, and allow direct naming of objects through pointing.

References

Altmann et al. (1988)

R.A. Altmann, A.N.Hawke and C.D.Marlin. "An Integrated Programming Environment based on Multiple Concurrent Views", *Australian Computer Journal*, 20(2), May 1988.

Baecker and Marcus (1986)

Ronald Baecker and Aaron Marcus. "Design Principles for the Enhanced Presentation of Computer Program Source Text", *Human Factors in Computing Systems: Proceedings of CHI '86*, April 1986, pp. 51–58.

Bauer and Wössner (1982)

Friedrich Bauer and Hans Wössner. *Algorithmic Language and Program Development*, Springer-Verlag, Berlin, 1982, pp. 195–213.

Bidoit and Choppy (1985)

Michel Bidoit and Christine Choppy. *ASSPEGIQUE: An Integrated Environment for Algebraic Specifications*, LNCS 185, Springer-Verlag, Berlin, 1985.

Bly and Rosenberg (1986)

Sara A. Bly and Jarrett K. Rosenberg. "A Comparison of Tiled and Overlapping Windows" *Human Factors in Computing Systems: Proceedings of CHI '86*, April 1986, pp. 101–106.

Böcker *et al.* (1986)

Heinz-Deiter Böcker, Gerhard Fischer and Helga Nieper. "The Enhancement of Understanding through Visual Representations", *Human Factors in Computing Systems: Proceedings of CHI '86*, April 1986, pp. 44–50.

Borning (1981)

Alan Borning. "The Programming Language Aspects of THINGLAB: a Constraint-Oriented Simulation Laboratory", *ACM Transactions on Programming Languages and Systems*, 3(4), October 1981, pp. 353–387.

Borning (1986)

Alan Borning. "Defining Constraints Graphically", *Human Factors in Computing Systems: Proceedings of CHI '86*, April 1986, pp. 137–143.

Brooks (1987)

Frederick P. Brooks. "No Silver Bullet: Essence and Accidents of Software Engineering", *IEEE Computer* 20(4), April 1987, pp. 10–19.

Brooks (1988)

Frederick P. Brooks. "Grasping Reality through Illusion",
Proceedings of CHI '88, May 1988, pp. 1–11.

Brown and Sedgewick (1984)

M.H. Brown and R. Sedgewick. "A System for Algorithm Animation", *Computer Graphics: Proceedings of SIGGRAPH '84*, **18**(3), July 1984, pp. 177–186.

Brown *et al.* (1985)

Gretchen P. Brown, Richard Carling, Christopher Herot, David Kramlich and Paul Souza. "Program Visualization: Graphical Support for Software Development",
IEEE Computer, August 1985, **18**(8), pp. 27–35.

Bundy (1981)

Alan Bundy. *Artificial Mathematicians: The Computational Modelling of Mathematical Reasoning*, Occasional Paper Number 24, 1981, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland.

Cardelli and Wegner (1985)

Luca Cardelli and Peter Wegner. "On Understanding Types, Data Abstraction, and Polymorphism", *Computing Surveys*, **17**(4), December 1985, pp. 471–522.

Carroll *et al.* (1988)

John Carroll, Penny Smith-Kerker, James Ford and Sandra Mazur-Rimetz. "The Minimal Manual", *Human-Computer Interaction*, **3**, 1987–1988, pp. 123–153.

Christensen (1968)

Carlos Christensen. "An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language", in *Interactive Systems for Experimental Applied Mathematics*, Melvin Klerer and Juris Reinfields (Editors), Academic Press, New York, 1968, pp. 423–435.

Cleaveland (1986)

J. Craig Cleaveland. *An Introduction to Data Types*,
Addison-Wesley, Reading, Massachusetts, 1986.

Cockshott (1989)

Tunde Cockshott. Personal communication, 1989.

Cockton (1987)

Gilbert Cockton. "A New Model for Separable Interactive Systems" in
Proceedings of Interact '87, Bullinger and Shacel (Editors), 1987, pp. 1033–1038

Cohen *et al.* (1986)

B. Cohen, W.T. Harwood and M.I. Jackson. *The Specification of Complex Systems*, Addison-Wesley, Wokingham, 1986.

Cox (1986)

Brad Cox. *Object-Oriented Programming*, Addison-Wesley, Massachusetts, 1986.

Cunningham (1985)

Ward Cunningham. *The Construction of Smalltalk-80 Applications*, Tektronix Computer Research Laboratory, Oregon, Draft copy, November 1985.

Dahl and Nygaard (1966)

Ole-Johan Dahl and Kristen Nygaard. "SIMULA – an ALGOL-based Simulation Language", *Communications of the ACM*, **9**(9), pp. 671–678, September 1966.

Danforth and Tomlinson (1988)

Scott Danforth and Chris Tomlinson. "Type Theories and Object-Oriented Programming", *ACM Computing Surveys*, **20**(1), March 1988, pp. 29–72.

Davis (1974)

Philip J. Davis. "Visual Geometry, Computer Graphics and Theorems of Perceived Type", *Proceedings of Symposia in Applied Mathematics* **20**, 1974, pp. 113–127.

Davis and Anderson (1979)

P.J. Davis and J.A. Anderson. "Non-analytic Aspects of Mathematics and their Implication on Research and Education", *SIAM Review*, **21**(1), January 1979, pp. 112–127.

Degano and Sandewall (1983)

Pierpaolo Degano and Erik Sandewall (Editors). *Integrated Interactive Computing Systems*, North-Holland, Amsterdam, 1983.

Duce and Fielding (1987)

D.A. Duce and E.V.C. Fielding. "Formal Specification – A Comparison of Two Techniques", *The Computer Journal* **30**(4), April 1987, pp. 316–327.

Duisberg (1986)

Robert A. Duisberg. *Animated Graphical Interfaces Using Temporal Constraints.* Tektronix Computer Laboratory Technical Report CR-86-05, January 1986.

Edel (1988)

Mark Edel. "The Tinkertoy Graphical Programming Environment", *IEEE Transactions on Software Engineering* **14**(8), August 1988, pp. 1110–1115.

Ehrig and Mahr (1985)

H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, Springer Verlag, Berlin, 1985.

Eisenstadt and Brayshaw (1987)

Marc Eisenstadt and Mike Brayshaw. *The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming*, Human Cognition Research Laboratory Research Report 21a, The Open University, October 1987.

Eisenstadt and Brayshaw (1989)

Marc Eisenstadt and Mike Brayshaw. *AORTA Diagrams as an Aid to Visualising the Execution of Prolog Programs*, in Kilgour and Earnshaw (1989), pp. 27–45.

Finzer and Gould (1984)

Laura Gould and William Finzer. *Programming by Rehearsal*, XEROX PARC Technical Report SCL-84-1, May 1984.

Furnas (1986)

George W. Furnas. "Generalized Fisheye Views", *Human Factors in Computing Systems: Proceedings of CHI '86*, April 1986, pp. 16–23.

Futatsugi *et al.* (1985)

Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud and José Meseguer. "Principles of OBJ2", *Proceedings of the 12th Annual Symposium on the Principles of Programming Languages*, New Orleans, January 1985, pp. 52–66.

Giannotti (1987)

Elena I. Giannotti. "Algorithm Animator: A Tool for Programming Learning", Proceedings of Eighth Technical Symposium on Computer Science Education, *ACM SIGCSE Bulletin* 19(1), February 1987, pp. 308–314.

Glinert (1987)

Ephraim Glinert. "Out of Flatland: Towards 3-D Visual Programming", in *Exploring Technology: Today and Tomorrow*, Proceedings of the Fall Joint Computer Conference, October 1987, pp. 292–299.

Glinert and Tanimoto (1984)

Ephraim Glinert and Steven Tanimoto. "PICT: An Interactive Graphical Programming Environment", *IEEE Computer*, 17(11), November 1984, pp. 7–25.

Goguen (1986)

Joseph A. Goguen. "Programming by Generic Example", *Proceedings of the Graph Reduction Workshop*, Santa Fe, September 1986. (Springer-Verlag LNCS 279, Fasel and Keller, Eds.).

Goguen *et al.* (1978)

J. A. Goguen, J.W. Thatcher and E.G. Wagner. "*An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types*", in Yeh (1978) pp. 80–149.

Goldberg (1984)

Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Massachusetts, 1984.

Goldberg and Robson (1983)

Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Massachusetts, 1983.

Gorny and Tauber (1987)

Peter Gorny and Michael J. Tauber (Editors). *Visualization in Programming*, Addison–Wesley, Reading, Massachusetts, 1987.

Grafton (1986)

Robert B. Grafton. "*Computing with Objects: a New Dimension in Visual Programming*", Proceedings of IEEE Computer Software and Applications Conference, October 1986, Chicago, Illinois, p.405.

GUCHCI (1988)

Practical Methods for Evaluating Human–Computer Interfaces, Glasgow HCI Group, Evaluating User Interfaces Course Notes, 3 & 4 October 1988.

Guttag (1975)

John V. Guttag. *The Specification and Application to Programming of Abstract Data Types*, Toronto University Computer Systems Research Group, PhD Dissertation, September 1975, Technical Report CSRG-59.

Guttag *et al.* (1978a)

John Guttag, Ellis Horowitz and David Musser. "Abstract Data Types and Software Validation", *Communications of the ACM*, 21(12), December 1978, pp. 1048–1064.

Guttag *et al.* (1978b)

John V. Guttag, Ellis Horowitz and David Musser. *The Design of Data Type Specifications*, In Yeh (1978) pp. 60–79.

Hadamard (1945)

Jacques Hadamard. *The Psychology of Invention in the Mathematical Field*, Princeton University Press, New Jersey, 1945.

Halbert (1984)

Daniel C. Halbert. *Programming by Example*, Xerox Technical Report CSD-T8402, December 1984.

Hansen (1971)

Wilfred J. Hansen. "User Engineering Principles for Interactive Systems", *Proceedings of the FJCC*, 1971, AFIPS 39, pp. 523–532.

Hayes (1987)

Ian Hayes (Editor). *Specification Case Studies*, Prentice-Hall, London, 1987.

Hoare (1982)

C.A.R. Hoare. "*Programming is an Engineering Profession*", Technical Monograph PRG-27, Oxford University Computing Laboratory, Programming Research Group, Oxford, May 1982.

Hoffmann and O'Donnell (1982)

Christoph M. Hoffmann and Michael J. O'Donnell. "Programming with Equations", *ACM Transactions on Programming Languages and Systems*, January 1982, 4(1), pp. 83–112.

Horowitz and Sahni (1976)

Ellis Horowitz and Sartaj Sahni. *Fundamentals of Data Structures*, Pitman Publishing, London, 1976.

Hutchins et al. (1986)

Edwin L. Hutchins, James D. Hollan and Donald A. Norman. *Direct Manipulation Interfaces*, in Norman and Draper (1986), pp. 87–124.

Ingalls (1981)

Dan Ingalls. "Design Principles behind Smalltalk", *Byte*, August 1981.

Johnson (1982)

D.S. Johnson. "The NP-Completeness Column: an On-going Guide", *Journal of Algorithms*, 3, pp. 89–99, 1982.

Johnson (1986)

Ralph E. Johnson. "Type-Checking Smalltalk", *Proceedings of OOPSLA '86*, September 1986, pp. 315–321.

Kilgour and Earnshaw (1989)

A. Kilgour and R. Earnshaw (Editors). *Graphical Tools for Software Engineering*, Cambridge University Press, 1989.

Liskov and Guttag (1986)

Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*, MIT Press and McGraw-Hill, 1986.

Liskov and Zilles (1974)

Barbara Liskov and Stephen Zilles. "Programming with Abstract Data Types", In Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages, SIGPLAN Notices 9(4), April 1974, pp. 50–59.

Mallgren (1982)

William R. Mallgren. "Formal Specification of Graphic Data Types", *ACM Transactions on Programming Languages and Systems*, 4(4), October 1982, pp. 687–710.

Miller (1956)

George Miller. "The Magical Number Seven, Plus or Minus Two: Some Limits on our Capability for Processing Information", *Psychological Review*, 63, 1956, pp. 81–97.

Moriconi and Hare (1985)

Mark Moriconi and Dwight F. Hare. "Visualizing Program Designs Through PegaSys", *IEEE Computer*, August 1985, 18(8), pp. 72–85.

Morris (1973)

James H. Morris. "Types are not Sets", *ACM Symposium on the Principles of Programming Languages*, October 1973, pp. 120–124.

Myers (1983)

Brad Myers. "INCENSE: A System for Displaying Data Structures", *Computer Graphics: SIGGRAPH '83 Conference Proceedings*, 17(3), July 1983, pp. 115–125.

Myers (1985)

Brad Myers. "The Importance of Percent-Done Indicators for Computer-Human Interfaces", *Proceedings of CHI '85*, April 1985, pp. 11–17.

Myers (1987)

Brad Myers. "Creating Interaction Techniques by Demonstration", *IEEE Computer Graphics and Applications*, 7(9), September 1987, pp. 51–60.

Myers (1989)

Brad Myers. *The State of the Art in Visual Programming and Program Visualization*, in Kilgour and Earnshaw (1989), pp. 3–26.

Nielsen (1988)

Jakob Nielsen. *Evaluating the Think Aloud Technique for use by Computer Scientists*, Presented at IFIP Working group 8.1 International Workshop on Human Factors of Information Systems Analysis and Design, London, July 1988.

Norman and Draper (1986)

Donald A. Norman and Stephen W. Draper (Editors). *User Centered System Design*, Lawrence Erlbaum Associates, New Jersey, 1986.

Papert (1980)

Seymour Papert. *Mindstorms – Children, Computers and Powerful Ideas.* Harvester Press, Brighton, Sussex, 1980.

Pong and Ng (1983)

M.C. Pong and N. Ng. "PIGS: A System for Programming with Interactive Graphical Support", *Software Practice and Experience*, 13(9), September 1983, pp. 847–855.

Powell (1987)

M.S. Powell. "Strongly-typed User Interfaces in an Abstract Data Store", *Software Practice and Experience*, 17(4), April 1987, pp, 241–266.

Powell and Linton (1983)

Michael L. Powell and Mark A. Linton. "Visual Abstraction in an Interactive Programming Environment", *Sigplan Notices*, 18(6), June 1983, pp. 14–21.

Raeder (1985)

Georg Raeder. "A Survey of Current Graphical Programming Techniques", *IEEE Computer*, August 1985, 18(8), pp. 11–25.

Reid (1989)

Peter Reid. *Dynamic Interactive Display of Complex Data Structures*, in Kilgour and Earnshaw (1989), pp. 60–70.

Reiss (1984)

Steven Reiss. "Graphical Program Development with PECAN Program Development Systems", *Sigplan Notices* 19(5), May 1984, pp. 30–41.

Reiss (1987)

Steven Reiss. "Visual Languages and the GARDEN System",
in Gorny and Tauber (1987), pp. 178–198.

Rubin *et al.* (1985)

Robert V. Ruben, Eric J. Golin and Steven P. Reiss. "ThinkPad: A Graphical System for Programming by Demonstration", *IEEE Software*, 2(2), March 1985, pp. 73–78.

Sheil (1981)

B. A. Sheil. "The Psychological Study of Programming",
Computing Surveys, March 1981, 13(1), pp. 101–121.

Shneiderman (1982)

Ben Shneiderman. "Designing Computer System Messages",
Communications of the ACM, 25(9), September 1982, pp. 610–611.

Shneiderman (1983)

Ben Shneiderman. "Direct Manipulation: A Step Beyond Programming Languages",
IEEE Computer, 16(8), August 1983, pp. 57–69.

Shneiderman (1986)

Ben Shneiderman. *Designing the User Interface*,
Addison-Wesley, Reading, Massachusetts, 1986.

Smith (1977)

David C. Smith. *PYGMALION: A Computer Program to Model and Stimulate Creative Thought*, Birkhäuser Verlag, Boston, 1977.

Smith (1987)

Randall B. Smith. "Experiences with the Alternate Reality Kit: an Example of the Tension between Literalism and Magic", *Human Factors in Computing Systems: CHI+GI 1987 Conference Proceedings*, April 1987, pp. 61–67.

Smith et al. (1983)

David C. Smith, Charles Irby, Ralph Kimball, Bill Verplank and Eric Herslem.
"Designing the Star User Interface", in Degano and Sandewall (1983), pp. 297–313.

Sommerville (1989)

Ian Sommerville. *Software Engineering*, (Third Edition)
Addison-Wesley, Wokingham, England, 1989.

Spector and Gifford (1984)

Alfred Spector and David Gifford. "The Space Shuttle Primary Computer System", *Communications of the ACM*, 27(9), September 1984, pp. 874–900.

Sutherland (1966)

William R. Sutherland. *On-line Graphical Specification of Computer Procedures.* MIT PhD Thesis, Lincoln Labs. Technical Report TR-405, May 1966.

Szekely (1987)

Pedro Szekely. "Modular Implementation of Presentations", *Human Factors in Computing Systems: Proceedings of CHI+GI 1987*, April 1987, pp. 235–240.

Thatcher *et al.* (1982)

J.W. Thatcher, E.G. Wagner and J.B. Wright. "Data Type Specification: Parameterization and the Power of Specification Techniques", *ACM Transactions on Programming Languages and Systems*, 4(4), October 1982, pp. 711–732.

Thiel (1984)

Jean Jacques Thiel. "Stop Losing Sleep Over Incomplete Data Type Specifications", *Proceedings of the Symposium on Principles of Programming Languages*, 1984, pp. 76–82.

Williams (1984)

Gregg Williams. "The Apple Macintosh Computer", *Byte*, 9(2), February 1984, pp. 30–54.

Yeh (1978)

Raymond Yeh (Editor). *Current Trends in Programming Methodology (Volume IV: Data Structuring)*, Prentice–Hall, New Jersey, 1978.

Zhong *et al.* (1988)

YouLiang Zhong, Seirei Ishizuka and Ryuichi Enari. "Integrating Abstract Data Types with Object-Oriented Programming by Specification-based Approach.", *Proceedings of the International Conference on Computer Languages*, Miami Beach, Florida, October 1988, pp. 202–209.

Appendix A
Glossary of Abbreviations

The following abbreviations and acronyms are used in the body of the thesis. Although they are always defined at their first occurrence, they have been gathered here for ease of referral. These abbreviations include those used in the references.

ACM	Association for Computing Machinery.
ADT	Abstract Data Type.
AFIPS	American Federation of Information-Processing Societies.
ARK	Alternate Reality Kit (c.f. Smith 1977).
AST	Abstract Syntax Tree.
BNF	Backus–Naur Format (for grammar descriptions).
FJCC	Fall Joint Computer Conference.
GPE	Graphical Program Editor (c.f. Sutherland 1966).
IEEE	Institute of Electrical and Electronic Engineers.
IFIPS	International Federation of Information-Processing Societies.
LC	Linkage Component.
LNCS	Springer-Verlag Lecture Notes in Computer Science Series.
MVC	Model-View-Controller user interface framework (of Smalltalk).
OOP	Object Oriented Programming.
PBE	Programming By Example.
PV	Program Visualisation.
SIAM	Society of Industrial and Applied Mathematics.
SIGCHI	ACM Special Interest Group in Computer-Human Interaction.
TPM	Transparent Prolog Machine (c.f. Eisenstadt and Brayshaw 1987).
UHLL	Ultra-High Level Language.
VP	Visual Programming.
UI	User Interface.

Appendix B
Specification Language Grammar

B.1 Introduction

The specification language supported by VISAGE has a conventional equational form similar to the numerous dialects that appear in the literature, e.g. Goguen et al. (1978) or Hoffmann and O'Donnell (1982). As it would only be used for the specification of relatively simple ADTs, there was no need to provide the sophisticated facilities of more advanced languages such as OBJ2 described in Futatsugi et al. (1985). Indeed, the minutiae of providing such sophistication would have obscured the discussion of the visualisation aspects of the research.

B.2 Notation

In the following Bachus-Naur description of the specification language grammar, the following notational conventions were used. Non-terminal symbols are shown in small capitals e.g. IDENTIFIER. Terminal symbols (literals) are shown as outlined text e.g. DATATYPE:. The following meta-characters are used:

|	denotes alternatives e.g. A | B means an A or a B is valid.
[...]	denotes that the part enclosed by the brackets is optional.
[...]*	denotes that the part enclosed by the brackets can occur zero or more times.
[...]+	denotes that the part enclosed by the brackets can occur one or more times.
::=	Separates a non-terminal from its expansion.
..	Denotes a range of terminals and is used to simplify the grammar, e.g. a .. z denotes the range of lower-case letters.

B.3 Grammar

SPECIFICATION ::= HEADER [ENRICHMENT] [USAGE] [COMMENT]
[SIGNATURE]* EQUATIONS

HEADER ::= **DATATYPE:** TYPENAME [[TYPENAME]]

ENRICHMENT ::= **ENRICHES:** TYPENAME

USAGE := **USES:** [TYPENAME]+ ;

COMMENT ::= " TEXT "

SIGNATURE ::= IDENTIFIER : FUNCTIONALITY [COMMENT]

FUNCTIONALITY ::= [DOMAIN] → RANGE

DOMAIN ::= TYPENAME [× TYPENAME]*

RANGE ::= TYPENAME

EQUATIONS ::= **EQUATIONS:** [EQUATION]* **ENDSPEC**

EQUATION ::= TERM = COMPLEXTERM [COMMENT]

TERM ::= SIMPLETERM | **ERROR!** | **UNKNOWN!**

SIMPLETERM ::= IDENTIFIER [ARGUMENTLIST]

TYPENAME ::= IDENTIFIER

ARGUMENTLIST ::= (TERM [, TERM]*)

COMPLEXTERM ::= CONDITIONAL | TERM

CONDITIONAL ::= IF TERM THEN TERM [ELSE COMPLEXTERM]

IDENTIFIER ::= LETTER [CHARACTER]*

LETTER ::= a .. z | A .. Z

CHARACTER ::= LETTER | 0 .. 9 | ! | ? | . | -

Appendix C

The stack Abstract Data Type

This appendix gives, for ease of reference, the complete specification of the stack ADT that has been used as a source of examples throughout the thesis. The specification uses the grammar given in Appendix B.

```
Datatype:  stack [data] uses: boolean;
"A generic first-in-last-out storage structure."

new.stack:                    →  stack
push:          data × stack   →  stack
pop:           stack          →  stack
top:           stack          →  data
is.empty?:     stack          →  boolean

Equations:
is.empty?(new.stack) = true
is.empty?(push(a,s)) = false
pop(push(a,s)) = s
pop(new.stack) = new.stack
top(push(a,s)) = a
top(new.stack) = error!
EndSpec
```

Stacks allows data values to be stored and accessed in a first-in-last-out fashion. The specification has a data type parameter called data which can be instantiated by the client to any actual data type. Operations are provided for creating a new stack (`new.stack`); adding data elements to a stack (`push`); accessing the last data element to be stored (`top`); removing the last data element stored (`pop`); and checking whether a stack contains any data elements (`is.empty?`). It is explicitly stated that trying to access the last-stored data value of an empty stack is an error. Note that this specification does not regard trying to remove the last-stored element from an empty stack as being an error. This demonstrates the usefulness of formal specification: the client is left in no doubt about the behaviour of `pop` in this circumstance even if it may be contrary to intuition.

Appendix D

The VISAGE Tutorial

Visage is a collection of tools for manipulating algebraic specifications of abstract data types (ADTs). These tools allow the form and behaviour of existing specifications to be explored, and also allow the creation of new specifications, either from scratch or by modifying existing specifications. This tutorial will give a quick tour of the VISAGE Browser, Helper and Playpen. It is not possible in a tutorial such as this to cover all the features of VISAGE and many of the details of using the system are covered in an overview document. However, if the steps in the following exercise are followed carefully, there should be no need to refer to that document.

D.1 The Visage Display

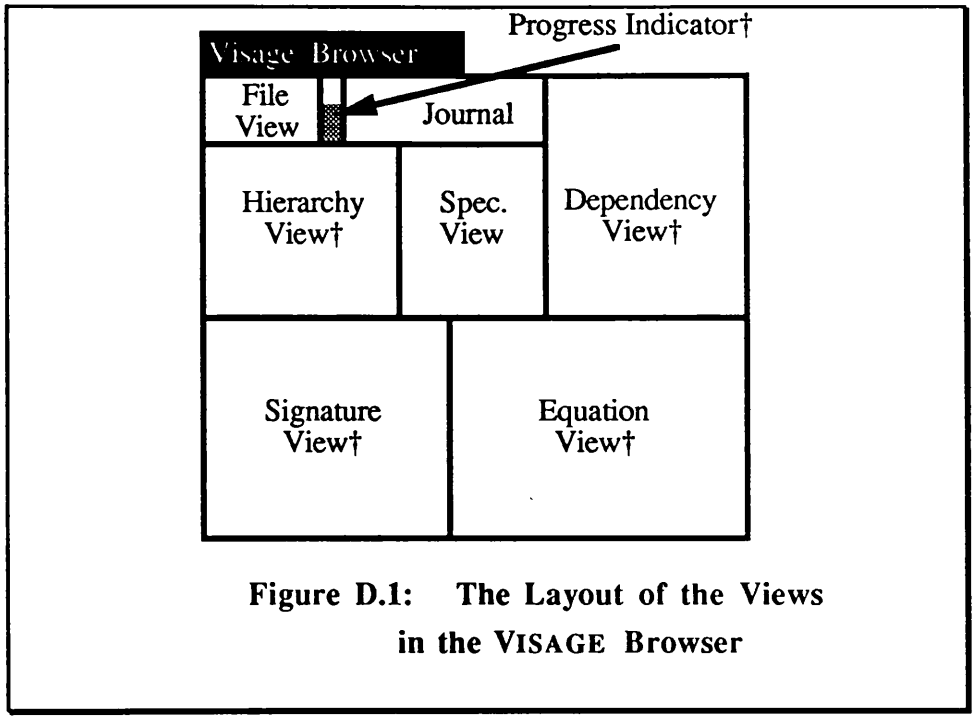


Figure D.1: The Layout of the Views in the VISAGE Browser

The VISAGE Browser appears in a separate window and has seven main views (the graphical ones are marked with a †). A schematic diagram of the display is shown in figure D.1 giving the relative position of each view. Each view has a label above it to help identify it. The purpose of each view should become apparent as we progress through the following exercise.

D.2 The Mouse and Menus

VISAGE uses a three-button mouse to point at and select objects graphically. Moving the mouse causes a cursor to move around the screen. This cursor usually looks like a small arrow but will sometimes change (usually to an hour-glass shape) to indicate when VISAGE is performing a lengthy or special operation.

Selecting Objects

The left mouse button is used to select or deselect objects currently pointed at by the cursor. When the cursor is over an icon, clicking the left mouse button will select it. A selected icon is highlighted by displaying its name in reverse video. Clicking on an icon that is already selected will deselect it. If an icon is shown *grayed-out* then it cannot be selected. Positions can also be selected by clicking the left mouse button when the cursor is at the desired location.

Choosing Commands from Menus

All the commands in VISAGE are held on menus which are simply lists of all the commands available in a certain context. To issue a command simply choose it from the menu. **A menu appears at the cursor location when the middle mouse button is pressed**†. To cancel a menu selection simply click the middle mouse button outside the menu area. The menu is different for each separate view within VISAGE and depends upon whether the cursor is over an object or not. In this tutorial, the menu that appears when the middle button is pressed with the cursor over the background of a graphical view is called the *Background Menu*. The menu produced when the middle button is pressed with the cursor over the arrowhead of a link is called the *Link Menu*. The menu produced when the cursor is over an icon is called the *Icon Menu*.

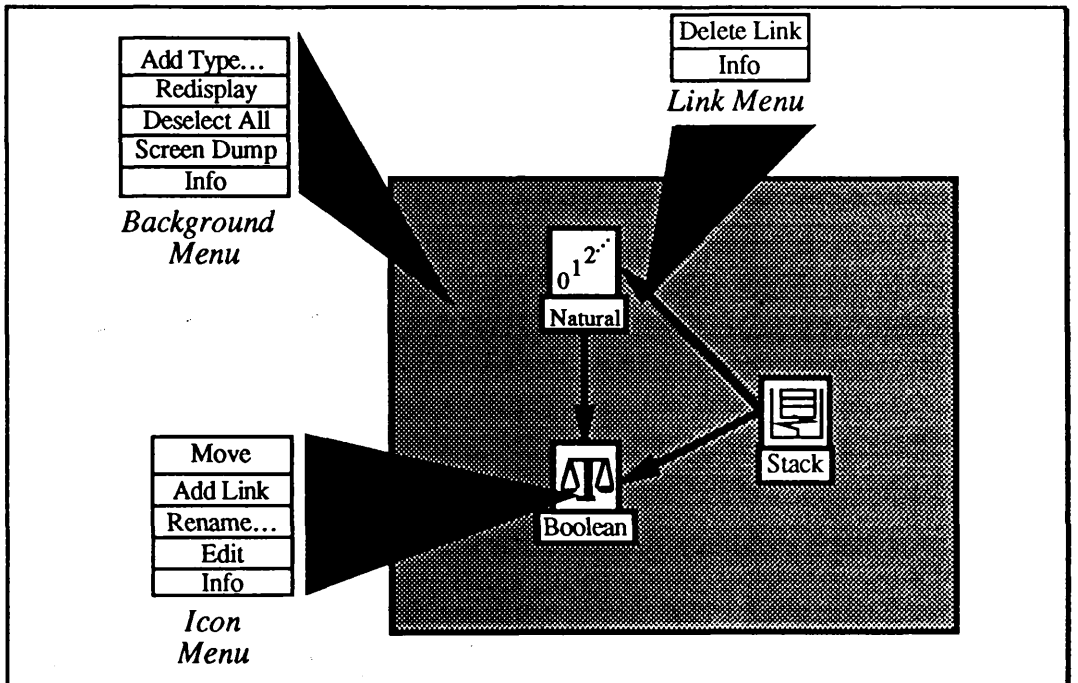


Figure D.2: Middle Mouse Button Menus in the Hierarchy View

† Menus only appear when a view contains some objects. There will be many opportunities to play with the mouse and its menus when tackling the exercise.

The right button has special menu commands but are only needed occasionally when using VISAGE and can be ignored for the moment: they will be described in detail whenever they are needed.

Figure D.2 shows examples of where the various menus appear when the middle mouse button is pressed within the view (in this case the Hierarchy View).

D.3 Introduction to the Exercise

The purpose of this exercise is to acquaint you with the main features of the VISAGE system. To help demonstrate its facilities we will develop and explore the specification of a **Stack of Natural Numbers**. Our specification needs several operations for creating, updating and accessing Stacks:

- **new.stack** creates a new (empty) stack for us to store the numbers.
- **push** puts a number on the top of a stack. The number and stack are given as parameters.
- **pop** removes the number from the top of the stack.
- **top** returns the number currently at the top of the stack.
- **is.empty?** returns a Boolean value saying whether the stack is empty or not.

Step 1. Creating a New Specification File

All specifications are held in files on the computer with each file containing perhaps many specifications. We will create a new file to hold our Stack specification. To do this use the CREATE NEW FILE menu command in the File View. A dialogue box will appear asking for a name for the new file. Type `tutorial` and press RETURN when you are finished (the DELETE key can be used to correct any mistakes). The box will disappear and after a little while the file name will appear in the File View. Select this (empty) file by clicking on its name: it will now appear in reverse video to show it has been selected. The Hierarchy View will now show two ADT icons named Boolean and Natural connected by a link. VISAGE supplies these automatically because they are used so frequently in writing other specifications (we will use them too). At this point you can play with the menus available within the Hierarchy View (see figure D.2).

Step 2. Creating the Stack Abstract Data Type

We start a new specification by adding a new ADT icon to the Hierarchy View. This is done by using the NEW TYPE command from the background menu of the Hierarchy View. Another dialogue box will ask for the name of the new type. Type `stack` and press RETURN, as before. A new icon will be attached to the cursor and can be moved anywhere within the view (VISAGE prevents you from placing an icon outside the view in which it belongs). Select the desired location (by clicking the left mouse button) to fix its location. (Note that in this case the icon's image has been pre-defined for you to look like a plate dispenser: an everyday "Stack"). It takes VISAGE a little while to create the new (empty) ADT but the thermometer-like progress indicator (see figure D.1) shows how things are progressing.

Step 3. Getting Information and Adding a Comment

The VISAGE Helper can automatically supply information about *any* object when you give the INFO command when over the object of interest. To see this information for our Stack ADT issue the INFO command when the cursor is over its icon in the Hierarchy View. A new window appears which you must position and size by pressing the left mouse button to fix its top-left corner, stretching the rectangle to the desired size and then releasing the left button. This window has two panes. The top one contains system-generated information that describes the object and cannot be edited. The bottom one is for user comments (and is initially blank). It is always useful in programming to have a short, informal comment about an object or operation. We can add one for our Stack by simply typing in the bottom pane and then issuing the ACCEPT command. Try this. When finished, the Helper can be removed by issuing the CLOSE command from the menu using the *right* mouse button. Try using the INFO command from the background menu to get a description of the view itself. Again, use the CLOSE command from the menu using the *right* mouse button when you are finished.

Step 4. Looking Inside the Stack

To "look inside" the Stack ADT we need to select its icon in the Hierarchy View. After a little wait, the other views will show various properties of the Stack. However, since the specification is currently empty, only the Specification View will have anything of real interest: it shows the template for the new Stack specification with the comment we added in the previous step. The Signature View will show the Stack icon on its own.

Step 5. Using Other ADTs

Our Stack needs two other ADTs in its definition: the Boolean and Natural ADTs (the former is needed by the `is.empty?` operation, the latter by the `top` and `push` operations). To allow the Stack to use these other ADTs, we add a *Usage Link* in the Hierarchy View from the Stack to each of them.

To create a usage link from the Stack to the Natural icon, move the cursor over the Stack icon and issue the ADD LINK icon menu command. A rubber-band line follows the cursor. Then select the Natural icon. A new link will be drawn. Note that the Specification View now has the line:

```
Uses: natural;
```

In addition, the Signature View now also has the icon for Natural (this means that we can now have operations that use Natural). Add a similar link between the Stack and Boolean icons. The Hierarchy View will now show something like the following figure (it is not important that the diagram look exactly the same, only that the arrows go in the proper direction):

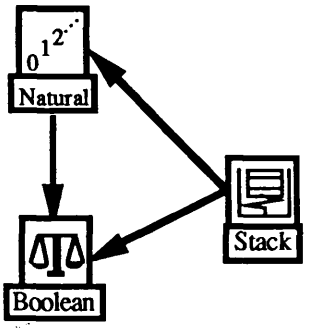
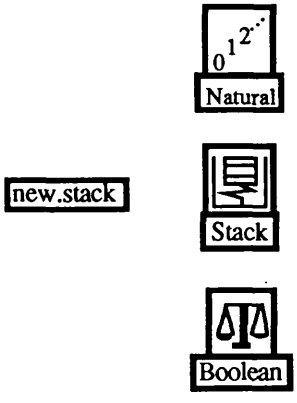


Figure D.3: Usage Links in the Hierarchy View

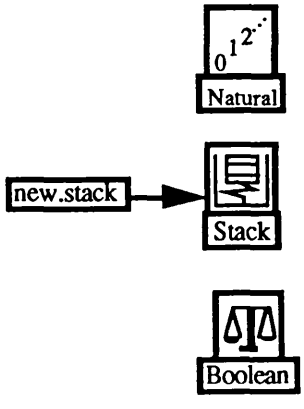
Step 6. Defining a New Operation

We are now ready to create the five Stack operations described in the introduction to this exercise. To demonstrate the definition of a new operation, the steps required to define the `new.stack` and `push` operations will be shown in detail in the sequence of figures below.



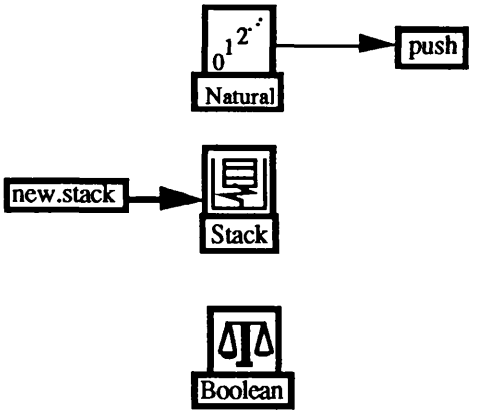
(i). A new operation is created using the NEW OPERATION command from the background menu of the Signature View. A dialogue box appears asking for the name of the new operation: type `new.stack` and press RETURN. An icon will be attached to the cursor; move it to a suitable position on the screen and click the left mouse button to fix it there. The Specification View should show the signature as:

```
new.stack: → unknown!
```



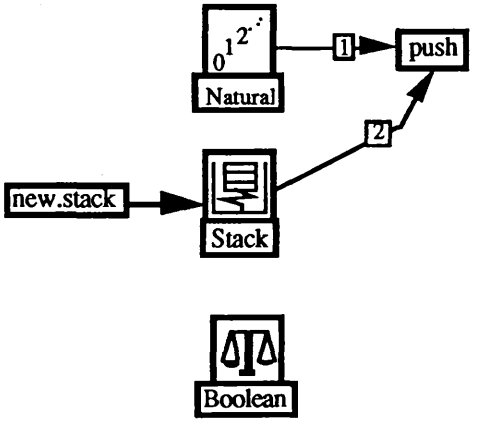
(ii). The `new.stack` operation produces a result of type Stack and we specify this by drawing a link *from* the `new.stack` icon *to* the Stack ADT icon. This is done by issuing the ADD LINK command while over the `new.stack` icon. A rubber-banded line follows the cursor and will disappear when you click a button when over the Stack icon. Since `new.stack` requires no arguments, this completes its definition. The Specification View should show the signature as:

```
new.stack: → stack
```



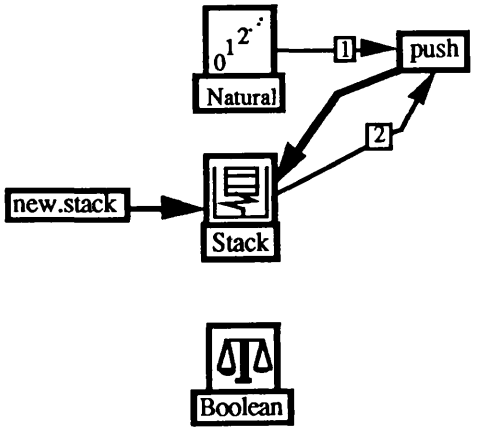
(iii). The `push` operation is more complex in that it has arguments. This figure shows the situation after creating the operation using NEW OPERATION and then adding its first argument link (done by drawing from the Natural icon to the `push` icon). The Specification View should show the signature as:

```
push: natural → unknown!
```



(iv). Here the second argument for `push` has been added by drawing a link from the Stack icon to the `push` icon. Notice that since there are now more than one argument link, they have been labelled to help distinguish them. The Specification View should show the signature as:

```
push: natural stack → unknown!
```



(v). The definition of `push` is completed by drawing its result link from the `push` icon to the Stack icon to show that `push` returns a Stack as its result. The Specification View should show the signature as:

```
push: natural stack → stack
```

Now try repeating the above process for yourself to define the `top`, `pop` and `is.empty?` operations so that they have the following signatures as shown in the Specification View:

```
top:        stack → natural
pop:        stack → stack
is.empty?:  stack → boolean
```

When complete, the Signature View should look something like figure D.4.

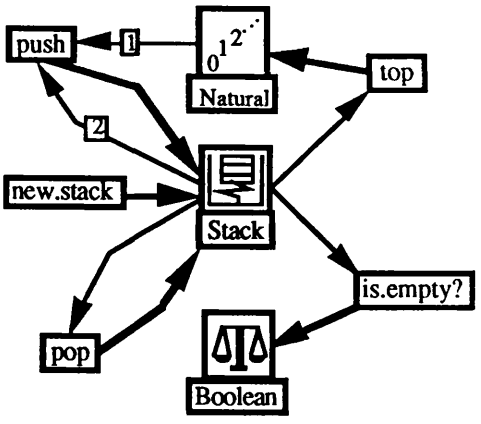


Figure D.4: Fully Defined Operations

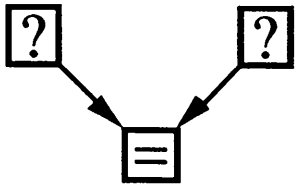
Step 7. Adding a New Equation

Now that the signatures of all the operations for the Stack ADT have been defined, we now need to give some equations that specify how they behave. The construction of two representative equations will be described in detail: the first states that a newly-created Stack is empty; the second states that pushing a number onto a Stack will create a Stack that is not empty, irrespective of what the initial Stack looked like. These are expressed algebraically (where a and s are Natural and Stack variables respectively) as:

1. `is.empty? (new.stack) = true`
2. `is.empty? (push (a,s)) = false`

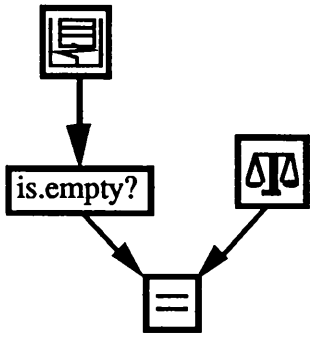
The following sequence of figures shows how these two equations are defined graphically. An equation is shown graphically as a tree with the root shown as a equals-sign icon. An equation is built by *plugging* operations and variables into *sockets* in the tree. *Every* icon (except the special equality one at the root) is a socket. A socket will only allow objects of the same type as itself to be plugged into it. **To plug an object into a socket, move the object (which is attached to the cursor) and select the socket icon.** When moving an icon, VISAGE flashes a socket if it is under the cursor and will allow the new object to be plugged into it. Plugging an object into a socket *replaces* any objects that were originally plugged in. If a new object cannot be plugged into a socket then it simply *falls off* the socket and lies on the surface of the view.

Follow the steps numbered (i) to (iv) to define the first equation.



(i). Create a new equation by issuing the Add NEW EQUATION background menu command in the Equation View. (A lot of things will happen on the screen while VISAGE creates the new equation: just ignore it for now). A graphical template tree for building the new equation is created. The two sockets here are special in that they will accept objects of *any* type to be plugged into them. A textual representation of the new equation is shown in the Specification View:

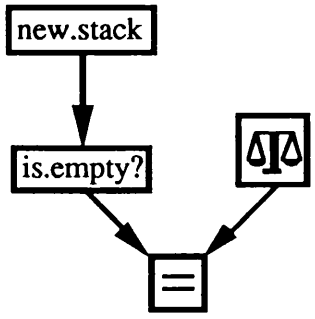
```
unknown! = unknown!
```



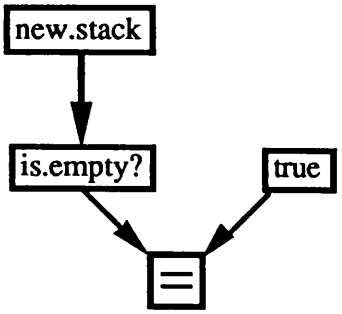
(ii). An instance of the `is.empty?` Stack operation is to be plugged into the left-hand socket. Create the instance by issuing the ADD SUBTERM command from the background menu in the Equation View. The next menu shows the options for the sub-terms. Select the *Operations* option by clicking on it. The next menu shows the different types of operations available: select the Boolean option (`is.empty?` returns a Boolean value). The final menu shows the available operations: select the `is.empty?`.

A new icon will be attached to the cursor ready for plugging into the left socket of the equation tree. (If you make a mistake in selecting the operation, simply click outside the menu area and start again). Once the `is.empty?` operation has been plugged in, the Equation View is updated to reflect the new connection: the right-hand side of the equation has the same type as the left (in this case Boolean) by changing the right-hand socket to only accept a Boolean term (the icon changes to show this). The `is.empty?` operation has introduced a new Stack-type socket corresponding to its argument. Note that the Specification View is *not* updated as you build the equation.

An alternative is to issue the ADD SUBTERM command from the *icon menu* when over the socket you want to plug something into. This also produces a menu from which you can select a constant, operation, etc. The difference is that the new subterm is constrained to be the same type as the socket. The new subterm is automatically plugged into the socket: the user does not have to place it.



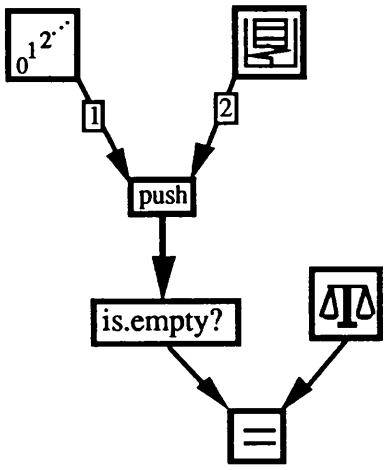
(iii). Select the `new.stack` Stack *constant* from the ADD SUBTERM background menu and plug it into the Stack-type socket corresponding to the argument of the `is.empty?` operation.



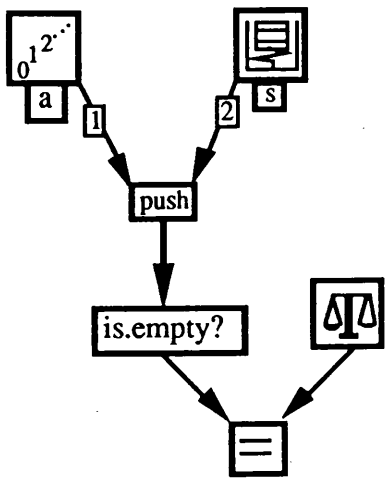
(iv). Select the Boolean *constant* `true` from the ADD SUBTERM background menu and plug it into the socket on the right-hand side of the equation tree. This completes the construction of the equation. Issuing the ACCEPT command from the background menu of the Equation View will incorporate this equation into the Specification View as: `is.empty?(new.stack)=true.`

When VISAGE incorporates the new equation into the specification the Specification View will highlight the new equation by emboldening it and the signatures of all the operations that it uses from the Stack ADT (in this case `is.empty?` and `new.stack)`. The Signature and Dependency Views also highlight the icons for these operations since the equation and its operations have been automatically selected.

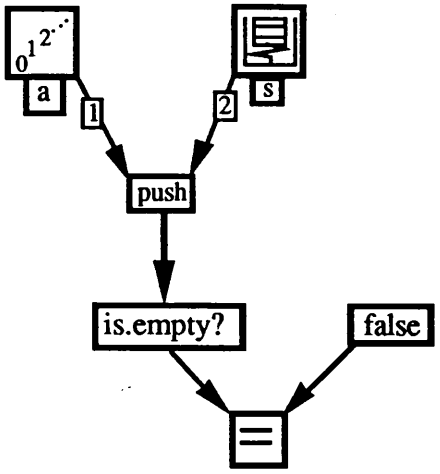
The second equation, `is.empty?(push(a,s))=false,` is constructed using the same techniques. Issue the ADD NEW EQUATION background command in the Equation View; perform steps (i) and (ii) from the previous page and then carry on with step (v) below.



(v). Select the `push` Stack operation from the ADD SUBTERM background menu and plug it into the Stack-type socket of the `is.empty?` operation. Since `push` has two arguments, it has two argument sockets: one of type Natural and one of type Stack.



(vi). Add two variables (called `a` (a Natural) and `s` (a Stack)) as the arguments to the `push` operation by using the *Variables* option of the ADD SUBTERM menu. A dialogue box appears asking for the names in each case.



(vii). Select the Boolean constant `false` from the ADD SUBTERM menu and plug it into the right-hand socket. This completes the construction of the equation. Issue the ACCEPT background menu command in the Equation View to incorporate this equation into the Specification View as:

```
is.empty?(push(a,s))=false.
```

This completes the graphical construction of the first two equations for our Stack specification. The other equations needed to complete the specification can be added by typing them directly into the Specification View†. This demonstrates the ability with VISAGE to use the textual and graphical views in a complementary fashion. To type in some text first select the place you want to start by clicking the left mouse button. This positions the text cursor: you can now type characters from the keyboard. The necessary equations are:

```
pop (push (a,s)) = s
top (push (a,s)) = a
pop (new.stack) = new.stack
```

Once they are all typed in correctly, issue the ACCEPT command in the Specification View. The graphical views will redraw themselves at this point in their attempt to produce an optimal display: the re-ordering of icons and links has no effect on the specification.

Step 8. Saving the Specification

Once we have a specification that we are happy with, we should save it for later use. By issuing the SAVE command from the File View menu, our new specification will be written to a file. A dialogue box asks for the name of the file to be used with the default being the name given when the file was created. In this case simply press RETURN. When the RETURN key is pressed, VISAGE will spend several seconds writing the specification to the file: the progress indicator shows how this is progressing.

Step 9. Testing the Specification: the Playpen

Once we have a specification it is desirable to see whether it conforms to our intuition about how it should behave. The VISAGE Playpen allows us to test our specification using example test data. The Playpen is started by issuing the PLAYPEN command in the File View. The Playpen appears in its own separate window and is shown in figure D.5 below. To position the window press the left mouse button down to

---

† Alternatively, you could define them graphically using the techniques just described, to gain more experience of this mechanism.

select the top-left corner of the window, drag the bottom-left corner and release the mouse button when the desired size is reached.

In the Playpen you can create example data as trees similar to the ones in the Equation View (the difference is that they don't have the equality icon at the root, and they *grow* downwards). You build these trees using the same technique of plugging objects into the tree by moving them over sockets and clicking.

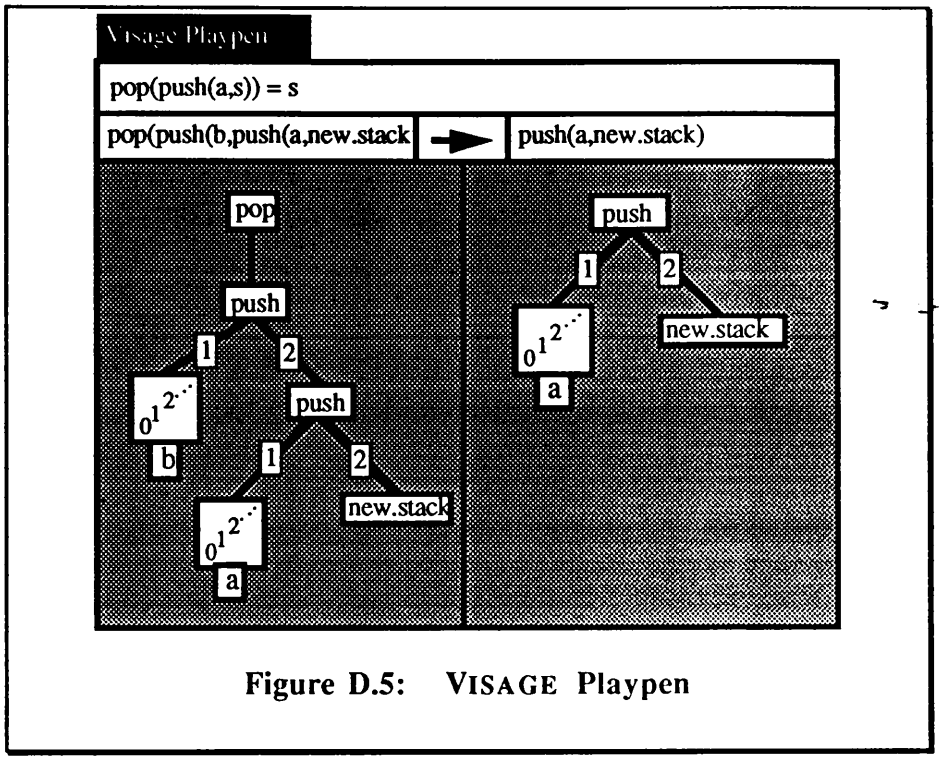


Figure D.5: VISAGE Playpen

For example, in figure D.5, the user has built the term

```
pop(push(b,push(a,new.stack)))
```

in the left-hand graphical view (although either view could have been used as there is no significance). The textual equivalent of the current graphical representation is always displayed in the text view immediately above each graphical view. When the REWRITE command is issued in the left graphical view, VISAGE scans the specification of Stack (since the term is of that type) looking for an equation to rewrite the term. If one is found it is displayed in the topmost text view: in this case it is `pop(push(a,s)) = s`. The result of rewriting the above term is then displayed in the right-hand view: `push(a,new.stack)`. The part of the term that has been rewritten is highlighted both graphically and textually in both halves of the Playpen so that you can compare the *before* and *after* states. The result of a REWRITE command can itself be rewritten or edited as usual.

For practise, try building (in either graphical view) the tree for the term:

```
is.empty?(pop(push(zero,new.stack))).
```

To clear the graphical view ready for building a term, issue the CANCEL background command. When you are finished issue the REWRITE command in the view in which you have been building. You will see that the term is rewritten to the new term:

`is.empty?(new.stack)` using the equation `pop(push(a,s)) = s`.

Compare the difference when the REWRITE ALL command is used instead. Experiment with other terms to see whether the Stack specification agrees with your intuition. You may find it useful to refer back to the complete specification given in the previous section of this tutorial.

To close the Playpen once you are finished experimenting, simply issue the CLOSE command from the right menu.

Step 10. At the End of the Exercise...

(Don't worry about just leaving the system when you are finished). This simple exercise should have given you some idea of the facilities available within VISAGE (and some of its shortcomings). To help improve VISAGE it would be very useful if you could use the system in a simple evaluation study: please see me for further details. Thank you very much for your time and patience.

Step 11. Further Exercises

If you would like to continue playing with the system using this exercise, the following short extensions to the exercise might be useful.

1. Try selecting and deselecting operation icons in the Dependency and Signature Views. What effect does this have on the other views? Try to select a particular equation in the specification using this technique.

2. Select the file `example` in the File View and explore the ADTs contained in it. In particular, try using all the menu commands in the Equation View to edit and generally manipulate equations.

Appendix E

Pre-Exercise Questionnaire

The purpose of this questionnaire is to determine your experience of using formal specifications and the particular computer system that will be used during the exercise. Any information supplied as part of this exercise (including the answers to questionnaires) will remain confidential. The sole purpose of the exercise is to evaluate the VISAGE system. When answering a question please circle the most appropriate answer from the statements below.

How much experience do you have of formal specifications of abstract data types?

1. I have no experience at all.
2. I have heard of them but have never written one.
3. I can understand simple ones but have never written one.
4. I have written specifications of simple ADTs (e.g. Stacks).
5. I have written specifications of more complex ADTs.

How much experience do you have of using a computer such as the Sun workstation?

1. I have never used such a computer.
2. I have used such a computer for less than a week.
3. I have used such a computer for more than a week but less than a month.
4. I have used such a computer irregularly for more than a month.
5. I have used such a computer regularly for more than a month.

How much experience do you have of using the Smalltalk system?

1. I have never used it.
2. I have used the Smalltalk Browser to look at classes and their methods, and know how to use the mouse and the pop-up menus.
3. I have added or modified methods belonging to a class.
4. I have added a class to the Smalltalk class hierarchy.
5. I have written an entire application using Smalltalk.

How confident do you feel that you could **understand** an existing specification of a simple ADT?

Not Very Confident 1 - 2 - 3 - 4 - 5 - 6 - 7 Very Confident

How confident do you feel that you could **write** a new specification of a simple ADT?

Not Very Confident 1 - 2 - 3 - 4 - 5 - 6 - 7 Very Confident

Appendix F

Evaluation Exercise

The purpose of this exercise is for you to use the VISAGE system on your own to investigate and extend the specification of a simple Abstract Data Type (ADT). This will draw on the techniques you encountered while using the Tutorial guide.

Purely for monitoring purposes, please enter below the time you start and finish the exercise. There is no time limit for this exercise.

Start Time		Finish Time	

The specification file `exercise` contains the specification for a Stack where the data values are Natural numbers. Select this file by clicking on it in the File View.

1. Add the signature of a new operation called `size?` to the Stack specification that given a Stack as its argument returns the number of data values that it contains.

2. Give a comment to the `size?` operation.

3. Add two equations to the specification to define the behaviour of the `size?` operation, constructing them graphically in the Equation View. Ensure `size?` performs correctly by trying it on some test data in the Visage Playpen.

4. Can you guess what the command MYSTERY does? Rename it to something that better fits its function. Use the Playpen to confirm your guess using some test data. For convenience, the equations involving `mystery` are reproduced below:

5.
```
mystery (new.stack, s) = s
mystery (push(a,s), t) = push(a, mystery(s,t))
```

6. Save your work by writing the specifications back to the file `exercise`.

Please fill in the time you finish the exercise. It would be greatly appreciated if you now complete the VISAGE questionnaire. Thank you.

Appendix G

Post-Exercise Questionnaire

The purpose of this questionnaire is to find out your reaction to using the Visage system. (Note that there are no *correct* answers). Please answer each question by circling the number on the scale that most closely fits your reaction. If you feel you need to qualify or expand an answer then please feel free to do so on the back of this sheet.

How confident do you feel that you could now use Visage to **understand** an existing specification of similar complexity to the Stack example?
Not Very Confident 1 - 2 - 3 - 4 - 5 - 6 - 7 Very Confident

How confident do you feel that you could now use Visage to **write** a new specification of similar complexity to the Stack example?
Not Very Confident 1 - 2 - 3 - 4 - 5 - 6 - 7 Very Confident

How useful did you find the VISAGE Playpen in understanding how an ADT behaved ?
Not Very Useful 1 - 2 - 3 - 4 - 5 - 6 - 7 Very Useful

How easy was it to follow what was happening on the screen?
Very Difficult 1 - 2 - 3 - 4 - 5 - 6 - 7 Very Easy

How did the speed of the system interfere with the task ?
Continuous Interference 1 - 2 - 3 - 4 - 5 - 6 - 7 No Interference

How much in control did you feel when using Visage?
Out of Control 1 - 2 - 3 - 4 - 5 - 6 - 7 Completely in Control

How well laid out and attractive was the Visage display?
Very Bad 1 - 2 - 3 - 4 - 5 - 6 - 7 Very Good

How much difficulty did you experience in selecting objects using the mouse?
No Problems 1 - 2 - 3 - 4 - 5 - 6 - 7 Lots of Problems

How much difficulty did you experience in using the pop-up menus?
No Problems 1 - 2 - 3 - 4 - 5 - 6 - 7 Lots of Problems

How helpful were the error messages produced by the system?
Very Poor 1 - 2 - 3 - 4 - 5 - 6 - 7 Very Good

How useful were the Progress Indicator and different cursor shapes in indicating the status of the system?
Not Very Useful 1 - 2 - 3 - 4 - 5 - 6 - 7 Very Useful

How quickly did you understand the use of the different views?
Very Slowly 1 - 2 - 3 - 4 - 5 - 6 - 7 Very Quickly

How much difficulty did you experience in building equations graphically?
No Problems 1 - 2 - 3 - 4 - 5 - 6 - 7 Lots of Problems

How helpful was the *Plugs and Sockets* metaphor in the graphical construction of equations and terms?
Not Very Helpful 1 - 2 - 3 - 4 - 5 - 6 - 7 Very Helpful

How useful was the visual feedback in showing type information when building equations and terms?
Not Very Useful 1 - 2 - 3 - 4 - 5 - 6 - 7 Very Useful

How often did you refer back to the Tutorial during the exercise?
Never 1 - 2 - 3 - 4 - 5 - 6 - 7 Constantly

What are your overall reactions to using VISAGE?		
Terrible	1 - 2 - 3 - 4 - 5 - 6 - 7	Wonderful
Frustrating	1 - 2 - 3 - 4 - 5 - 6 - 7	Satisfying
Dull	1 - 2 - 3 - 4 - 5 - 6 - 7	Stimulating
Difficult	1 - 2 - 3 - 4 - 5 - 6 - 7	Easy

What would you say was the worst aspect or part of VISAGE?

What would you say was the best aspect or part of VISAGE?

If you have any comments or suggestions about any aspect of the Visage system or this exercise, please feel free to write them down on the back of this questionnaire.

Thank you for your time and assistance.