AUTHOR: **Richard Ian Cartwright** DEGREE: **Ph.D.**

TITLE: **Geometric Aspects of Empirical Modelling: Issues in Design and Implementation**

DATE OF DEPOSIT: ....

Geometric Aspects of Empirical Modelling: Issues in Design and Implementation

by

Richard Ian Cartwright

Thesis

Submitted to the University of Warwick

for the degree of

Doctor of Philosophy

Department of Computer Science

September 1998

Dedicated to the memory of

Dominic John, 1973–1995

Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my friend and supervisor, Meurig Beynon, for his valued (or should that be "variabled") guidance through the research and writing of this thesis. His constant enthusiam for my work was tireless and inspirational.

I would also like to thank Helen Cartwright, my wife, for her support and tolerance of this work, as well as her proof reading — I'll never split an infinitive again! I am indebited to my empirical modelling co-conspirator and friend, Dominic Gehring, for his proof reading and all night companionship during the final phase of this work. Thanks are due to all the members of the empirical modelling project who have all been helpful and supportive of my work.

Special thanks go to all my family for the help and support that they have given me in so many ways.

Finally, thanks also goes to the EPSRC and Matra Datavision who funded the research through a CASE award studentship, and to Meurig Beynon for making it happen. I must also thank all my other friends and colleagues in the Department of Computer science for their support.

Declarations

This thesis is presented in accordance with the regulations for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work in this thesis has been undertaken by myself except where otherwise stated.

The various aspects concerning cognitive artefacts for timepieces have been published in [BC95]. The examples relating to higher-order definitions have previously appeared in [GYC$^+$96].

Abstract

Empirical modelling is a new approach to the construction of physical (typically computer-based) artefacts. Model construction proceeds in an open-ended and exploratory manner in association with the identification of observables, dependency and agency. Knowledge of the referent is acquired through experiment, and — through the use of metaphor — interaction with the artefact is contrived so as to resemble interaction with the referent. Previous research has demonstrated the potential for empirical modelling in many areas. These include concurrent engineering, virtual reality and reactive systems development.

This thesis examines the relationship between empirical modelling and geometric modelling on computer systems. Empirical modelling is suggested as complementary to variational and parametric modelling techniques commonly used in software packages for geometric modelling. Effective techniques for exploiting richer geometric models in visual metaphors within empirical modelling are also developed.

Technical issues arising from geometric aspects of existing empirical modelling tools and case-studies are reviewed. The aim is improve the efficiency of existing implementations, and to introduce data representations that better support geometric modelling. To achieve this, a mathematical model (the DM Model) for representing the dependency between observables is introduced, and this is used as the basis for a new algorithm for propagating updates through observables. A novel computing machine (the DAM Machine) that maintains dependencies representing indivisible relationships between words in computer store is derived from the DM Model. Examples of the use of this machine for the representation of geometry are presented. In implementation, a comparative efficiency gain is achieved by the DAM Machine over existing tools. This allows for the real-time animation of models.

A novel and general approach to the representation of data, suitable for integrating empirical modelling and general Java applications, with additional support for collaborative working, is developed. Object-oriented programming methods provide the foundation for new tools to support this representation. The *empirical world* class library allows a programmer to implement new applications for shape modelling that support empirical modelling and integrate a wide range of shape representations. A method of integrating these geometric techniques into spreadsheet-like environments that are well-adapted to support empirical modelling is proposed.

Abbreviations

API - Application Programming Interface

B-Rep - Boundary Representation

BCSO - Boolean Compound Soft Object

CAD - Computer-Aided Design

CADNO - Computer-Aided Design Notation

CAM - Computer-Aided Manufacturing

CSG - Constructive Solid Geometry

DAM - Definitive Assembly Maintainer

DM Model - Dependency Maintainer Model

HTML - Hypertext Mark-Up Language

JaM - Java Maintainer

VRML - Virtual Reality Modelling Language

Chapter 1

Introduction

Empirical modelling is concerned with the exploration of artefacts through experimentation and the construction of new instruments that support interactive elaboration of these artefacts. These artefacts inform the process of discovering new concepts through the use of metaphorical representations of state. By exploiting novel computational abstractions and through a broad concept of what constitutes a computer — any reliable, interpretable, state-changing device — empirical modelling can be used to create computer-based artefacts that explicitly imitate phenomena as observed. Artefact construction is similar to the process of developing an engineering prototype [Rus97]; it is open-ended and does not require early circumscription to a models components or parameters. Further exposition of empirical modelling principles in relation to education and learning can be found in [Bey97], and in relation to the foundations of artificial intelligence in [Bey98a].

Empirical modelling has the scope to support early conceptual development, concurrent engineering, reactive systems design, computer-aided design and software requirements capture. Geometry in its broadest sense plays an important role in establishing metaphors for the representation and communication of the states of artefacts. In this thesis, aspects of geometry that support metaphorical state representations are studied with reference to the principles of empirical modelling. This process necessarily involves the design and implementation of tools that allow empirical modelling to exploit richer geometry. These tools can provide appropriate and flexible interfaces for empirical modelling that support concurrent engineering, possibly on distributed computer systems, in an efficient manner. The process also involves the development of empirical modelling tools and techniques to support design. This requires re-appraisal of existing implementation techniques to better support script management, data structure, agent privileges and solid modelling.

The representation of artefacts benefits from strong visual metaphors that closely imitate their observed real world referent. Empirical modelling aspires to the use of photo-realistic images and virtual reality environments for the real-time animation of models, although empirical modelling on a computer system can be adequately carried out without these benefits.

1.1 Motivations for Empirical Modelling

Human agents play many different roles in design and the creative process. Some of these many different roles are categorised in the triangle in figure 1.1. In Section 1.1.1, an analogy with the composition and performance of music is drawn to illustrate how modern technology supports the conflation of these many roles. In this thesis, empirical modelling and geometric shape modelling are central themes. The processes of modelling shape and composing and performing music have a significant affinity as both involve establishing metaphors for aspects of the real world that have an analogue character. The figure refers to the roles of agents in the design of computer hardware and software systems, to assist in the explanation of the relationship between empirical modelling and the development and the use of computer systems in Section 1.1.2.

1.1.1 An Analogy from Music

In a traditional view of music, a *composer* writes music in a well-understood notation. This music is played by a *performer* who adds their own interpretation based on their insight and understanding of the composer's other work and general style. The performer (of non-vocal music) plays the music on an instrument created by an *instrument maker* who has assessed the available materials and used their knowledge of the construction of similar instruments to build a new one. The composer does not need to know about the construction process for the instrument or how to play music with it to write music for it. They only need a conception of the kind of sound that instruments make[1]. The performer does not need to know how to compose or about the internal structure within a composition be able to play it. Nor do they need to know how to construct the instrument they are playing.

The composer is creating recipes for interaction between performer and instruments from their conceptualisation of the interaction process. This process is prescribed to a certain extent by the musical notation that is used to communicate between the composer and the performer. The performer executes this interaction following the recipe to a certain degree and with an expertise on how to use their instrument within a musical framework, such as where to place their fingers on the string of a violin to achieve a particular note. Many instruments allow the user to play notes that cannot be recorded in conventional musical notation. A violinist can choose to place their finger half way in between two notes that can be scored and produce a quarter tone. This ability of the performer to play beyond anticipated limits is explored in music of the twentieth century, where composers have invented new notations for the communication of sound concepts such as quarter tones and notations that give the performer more freedom for improvisation[2].

[1] The British composer Vaughan-Williams wrote at least one concerto for every instrument of the orchestra, while only being able to play a few of them himself.

[2] A seminal composition that demonstrates the use of new notations for music is Penderecki's

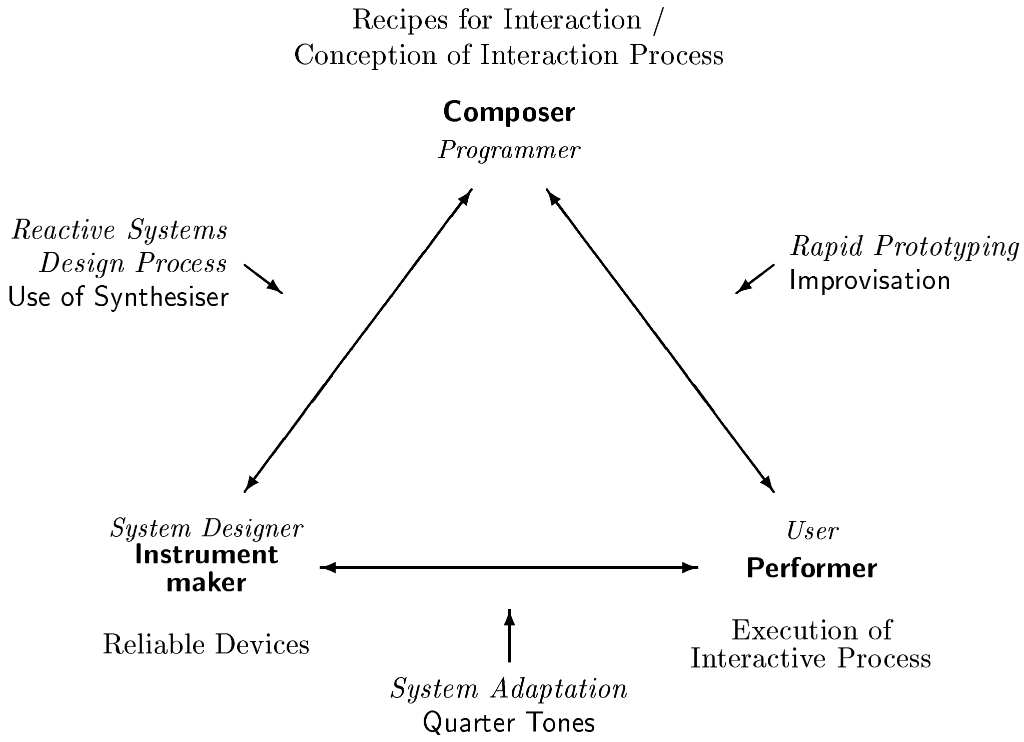


Figure 1.1: Roles in musical and *computer-based* interaction.

Technical developments have supported the conflation of the three musical roles, with significant benefits to all agents involved. The electronic synthesiser is a new musical instrument that allows a composer to experiment with new sounds that are beyond the scope of existing instruments, or can assist a performer to compose music by recording their improvisations and translating them into a musical notation via a computer system. Similar conflations of roles can benefit concurrent engineering, where there is a need for exploratory contexts that support references, values and privileges. Empirical modelling can support the conflation of these roles by providing interactive, computer-based exploratory contexts.

1.1.2 Computer-Based Interaction

In computer-based interaction, there are three conventional roles for human agents. The computer *systems* expert knows about the physical aspects of constructing reliable computational devices. The *programmer* (consider in the broad sense as an agent who specifies, designs, codes and tests software) creates recipes for interaction with the computer system. The interface between the programmer and the systems expert is through programming languages, libraries of code and common abstractions. The computer *user* uses applications created by programmers for their own purposes and has no interest in source code for the applications. All three roles must exist for the general use of computers in this conventional framework.

Empirical modelling on computer systems supports the conflation of the roles of programmer, user and systems expert by allowing them to explore common areas between their roles. In the development of *reactive systems*, an agent needs to have a good understanding of the reliable devices in the system to create software to control the system. This is the same for the evolving process of design, where a designer needs to understand the materials and environment that situate their model

Threnody for the Victims of Hiroshima.

to create new and improved designs. Alternatively, computer users may decide to use their computer hardware in a way that was not anticipated by the hardware designers, such as the use of video-conferencing equipment as security surveillance devices.

Suchman introduces the concept of *situated actions* in [Suc87]. She considers all purposeful actions as situated actions that take account of particular, concrete circumstance. She argues that no matter how carefully you make a plan before an event, it cannot take account of unforeseen circumstances. When asked to describe a plan in retrospect, plans filter out the detail that characterises situated actions. Situated actions are similar to the common understanding of music by the composer and the performer. A performer may adapt their performance to suit an audience or current stylistic fashion. In the same way, a user of a software package for computer aided design (CAD) may wish to explore the possibilities of shape beyond those permitted by the user interface of their software package[3] in the same way that they might use a paper sketch-pad.

1.2 Motivations for this Thesis

This thesis examines geometric aspects of empirical modelling from two angles:

1. *Empirical modelling for geometry*, where empirical modelling tools and techniques support geometric shape modelling, conceptual design and concurrent engineering;

2. *Empirical modelling with geometry*, where geometry supports the empirical modelling process in a non-geometric design context.

[3]The incorporation of a *Visual Basic* interpreter in the Microsoft *Excel* spreadsheet [Jac97, KDS96] is an example of a computer program that allows a user to become a programmer, tailoring the spreadsheet functions to their own needs.

The possibility of supporting both these angles through implementation on computer systems is examined in detail. In relation to geometric modelling, this thesis concerns the creation of instruments for geometric modelling that can support the development of computer-based applications for shape modelling and an exploratory design environment for a designer using a computer. Much of the discussion in the thesis can be applied generally to the implementation of tools that support empirical modelling, although the results presented are based on research relating to how to improve existing tools to better support geometry. Most of the case-studies presented relate to geometry and shape modelling.

Existing work on empirical modelling, and its application in a number of different case-studies[4], has demonstrated potential to support open-ended development and experimental interaction in several different contexts. In relation to supporting the empirical modelling angle, the work in this thesis is a continuation of existing research into the combination of empirical modelling and geometric modelling [ABH86, BC89, Car94a]. In the previous work, both existing tools that support empirical modelling and interfaces to CAD packages have been investigated. The aim was to bring the support for empirical with and for geometric modelling closer to the standard of interactivity and graphical flair that is expected by users of modern computer software. The existing tools proved too slow and it was found that the coding of CAD packages restricts the freedom of expression required to support open development.

This work is motivated by the need to overcome the technical hurdles described above and to be able to demonstrate the benefits of uniting empirical and geometric modelling. Empirical modelling is *observation-oriented*: all variables in empirical models are considered as representing the current state of some observed quantity in a referent. Empirical modelling requires strong visual metaphors and

[4]For a background to empirical modelling, see section 2.2 of chapter 2.

good interfaces for direct manipulation to represent entities as they are observed and controlled. This requires integral support in tools for the specification and manipulation of geometry. Empirical modelling takes direct account of dependency between observables in a manner that cannot be achieved by circumscribed mathematical models. In many geometric structures, there is inherent dependency (such as characteristic patterns of incidence, and dimensional constraints). For this reason, empirical modelling cannot make direct use of traditional mathematical models of geometry, and other representation techniques are required.

Section 1.2.1 introduces the essential concepts that underlie empirical modelling and the terminology relating to empirical modelling that are required in later sections. This is followed in section 1.2.2 by an explanation of the aims of this research work.

1.2.1 Essential Empirical Modelling Concepts

At this early point in the thesis, it is important to define some key concepts relating to Empirical Modelling related to the discussion throughout the document (see also [Tea, Bey85, BY88a, Bey97, Bey98a]). *Empirical modelling* focusses on the construction of artefacts (models) for interaction and experiment in a doamin that is not yet well understood. Analyis of a domain or referent proceeds through the identification of the fundamental structure of our personal experience of that domain. The atomic elements of this structure are *observables* which consist of unique name and value pairs. Each observable represents a measurable or quantifiable element of the domain and its current value in the model corresponds to a state of the referent. Empirical modelling principles concern the identification of measurable observables in some real-world referent and, if appropriate, the observation of dependency between observables and the agents associated with these observables.

Observables can be given a unique identifier and their current value can be expressed by the statement "*identifier* = *value*", in the manner of assignment to a variable in a conventional programming language. This statement is a *definition* of the value of the observable. An artefact constructed through empirical modelling will have a collection of many observables, which each have their own definition. Such a collection of definitions is known as a *definitive script*. A *definitive notation* consists of the data types and operators of the underlying algebra suitable for modelling the domain over which definitive scripts for models are constructed.

Analysis of the domain also involves the observation of the syncronised patterns of changes in observables. Observables that are seen to be dependent on other observables can also be expressed as definitions by describing the relationships between their values. These definitions are of the form "*identifier* **is** *expression*", for example "`width is 2*height`". If a value that forms the right-hand side of a definition is changed, the value of a dependent observable is updated consistent with its definition. The process is similar to interaction with a spreadsheet application, where a user identifies the relationships between cells on the spreadsheet and changing the value in one cell causes other dependent cells to update automatically.

The process of constructing computer-based artefacts using definitive scripts is called *definitive programming*. Support for definitive programming on computer systems is provided by the EDEN interpreter [YY88, Yun90], a generic evaluator for definitive notations. Some EDEN definitions are shown in Table 1.1, where the value of observable `a` is defined to be equal to the sum of the values of observables `b` and `c` (`9` in the example). Subsequent change to the value of `b` or `c` during interaction with the interpreter will cause the value of `a` to update. Interaction with EDEN models takes place through an ongoing process of typing definitions line-by-line on-the-fly. These definitions represent the introduction of new observables and the *redefinition* of existing ones.

The definition of observable `d` in Table 1.1 to depend on the values of `b`

EDEN definitions	EDEN functions and procedures (actions)

`b = 4;`
`c = 5;`
`a is b + c;`
`d is f(b, c);`
`e = a;`
	`func f {`
	`  return $1*$1 - $2;`
	`}`
	`proc p : c, d {`
	`  writeln(c, d);`
	`}`

Table 1.1: Example definitions, functions and procedures for EDEN.

and `c` by a user defined *function* `f`. An EDEN specification of the function `f`, where $\mathtt{f}(x, y) = x^2 - y$ is shown in the table. If the code describing a function is updated, then all values defined using the function are updated. EDEN also includes procedural triggered `actions`, like `proc p` in Table 1.1. These are segments of code that are executed every time the values of particular observables are updated. In the example, every time the value of `c` or `d` changes, the current values of both `c` and `d` are written out by the EDEN interpreter program[5].

1.2.2 Aims

Empirical modelling principles have not yet been applied to the specification of data types for underlying algebras for observables and their operators. In an ideal general definitive programming tool, it would be possible to introduce new data types on-the-fly to correspond with new observations. Using existing tools, a domain to be modelled must be understood in terms of the underlying algebra for a new definitive model in that domain prior to model construction. The efficiency of implementation of an ideal tool should be sufficient to provide realistic interaction corresponding to a user's experience of the domain, including domains containing geometric models

[5] Note that there is a distinction that exists *only* in the EDEN notation between the tokens "`is`", for a definition where a value is maintained automatically to be consistent with its definition, and "`=`", for a value that remains exactly the same until it is redefined. In the example, observable `e` remains equal to the value of observable `a` at the exact moment of definition.

with a large volumne of associated data. The existing tools provide no support for abstract data types, are not suitable for large volumes of data and execute slowly as they are based in interpreters. A central aim of this thesis is to investigate and experiment with other ways of implementing empirical modelling instruments to support generalised data representations and improve efficiency, so as to enhance the quality of stimulus-response interaction with tools.

This thesis also aims to address issues that are not tackled by existing tools such as EDEN, or at best are handled rather clumsily. These include:

- the representation and support for the privileges of interacting agents;
- unifying representations for references, values and privileges to support applications such as concurrent engineering;
- the potential for a greater degree of flexibility, efficiency and portability;
- the dynamic instantiation/elimination of definitions;
- support for higher-order definitions;
- integrated extension to general data types and operators beyond those currently supported.

The integration of support for general data structure and geometry will lead to richer computer-based models that correspond more closely to their real world counterparts. It will be possible to imitate geometric characteristics of the referent in the metaphor for interaction with an artefact. By improving the efficiency of implementation, it should be possible in the longer-term to produce convincing *real-time* animations and simulations of real-world reactive systems. A central goal of the work is to demonstrate the potential for the use of empirical modelling principles to support open-ended geometric modelling.

1.3 Contents of Thesis

In this section, a brief outline of the thesis is presented as a guide to the structure and dependency between the chapters and sections. The thesis describes a model for formulating and reasoning about dependency maintenance and includes information about three new tools developed as part of the research work. The technical issues raised by existing tools are identified, and a method for overcoming these is proposed. A new Java class library that allows a programmer to implement new applications for shape modelling and integrate a wide range of shape representations is presented. A method of integrating these geometric techniques into spreadsheet-like environments that are well-adapted to support empirical modelling is proposed. In section 1.3.2, the overall contribution of the thesis is described.

1.3.1 Thesis Layout

This thesis is divided into nine chapters, of which this is the first introductory chapter. Chapter 2 provides the background to this work, highlighting the relevant literature from geometric modelling as well as empirical modelling. This chapter also includes a discussion of the underlying concepts for empirical modelling with geometry to support the construction of computer-based artefacts, and empirical modelling to support the abstract, open development of geometric models. Case-studies to illustrate these concepts address modelling of timepieces and physical tables.

Chapter 3 examines the technical challenges of representing geometric data and dependency in definitive scripts, by examining existing definitive notations and other small case-studies. At the end of the chapter, the use of serialised data types to represent all types of data in a definitive notation is proposed and justified. Chapter 4 presents a method of reasoning about dependency maintenance free of concerns of data types and structure, called the *Dependency Maintainer Model* (DM Model). The new *block redefinition algorithm* that improves efficiency in dependency maintenance by considering several redefinitions in a block simultaneously is described with the DM Model. The chapter ends by examining the relationship between dependency maintenance and conventional algorithms, such as sorting.

Chapters 5 and 6 describe two programming toolkits based on the DM Model of Chapter 4 that both implement the block redefinition algorithm. Chapter 5 describes the design of a novel machine concept called the *Definitive Assembly Maintainer Machine* (DAM Machine), contrived to maintain dependencies between words of computer RAM store. This machine has been implemented over the ARM architecture [Fur96] and programmed to support the DoNaLD definitive notation for line drawing [ABH86]. This use of the toolkit is described and the possibility for efficient animation is demonstrated with a case-study of an engine model.

Chapter 6 presents the *Java Maintainer Machine* (JaM Machine) application programming interface (API), which maintains dependency between Java objects [CH96]. The toolkit offers support for multi-user collaborative working and distribution of scripts to several computer systems simultaneously over a TCP/IP network [Har97]. Chapter 7 and 8 describe *empirical worlds* — a case-study in the use of the JaM Machine API for shape modelling. Chapter 8 includes a description of the *empirical world builder* application that integrates the empirical world class library and supports shape modelling within a World-Wide-Web browser application. Virtual reality worlds (cf. VRML [ANM96]) are used as the display mechanism for geometry described in a definitive script.

The thesis ends in Chapter 9 by drawing conclusions from the research described. Further research work is proposed, including the possibility of a new style of application to support shape modelling that is based on spreadsheet ideas.

1.3.2 Contribution of Thesis

In respect of empirical modelling, the work in this thesis contributes a new method for handling data structure with dependency to better support observation-oriented modelling. The new *block redefinition algorithm* provides a more efficient means to propagate updates through the values of observables by considering more than one redefinition simultaneously. The JaM Machine API brings many of the features available in object-oriented programming to empirical modelling. The use of compiled code in JaM and assembly language directly in the DAM Machine allows for empirical modelling that supports the efficient and smooth animation of models. This also enhances the scope for interactive experimentation with a model that plays an essential role in its construction and interpretation. The use of the Java programming language provides a platform-independent way of integrating existing (not definitive) libraries of objects for graphical user interfaces, device control, networking and databases, into applications that support empirical modelling.

In respect of geometric modelling, the thesis presents a new approach to representing dependency in geometric modelling that allows for open development and supports conceptual design "*what-if?*" experimentation. This approach is different in character from variational and parametric modelling techniques in that it does not require the solution of several constraints. The use of implicit function representation for shape allows designers to explore many different representations for shape in a unified design environment. This is demonstrated by the proof-of-concept *empirical world builder* tool. Scripts of definitions that represent a geometric model can be shared between more than one workstation and there is support provision (by the JaM Machine API) for collaborative working and concurrent engineering.

For more general computer science, this thesis provides an insight into a new approach to the use of objects in object-oriented programming, where the communication between objects is through dependency-maintenance mechanisms. Insight into how the integration of general data structure and dependency can be achieved in the same definitive script provides the possibility to use empirical modelling in a broader range of applications. These definitive scripts can support the incremental construction and open-ended development of models over a diverse and extendable range of data types. This could provide a basis for new powerful spreadsheets where the values in cells can be richer data types than text, numbers and dates alone. It also provides support for software development and modelling using computer systems in which the roles of the user, programmer and computer systems designer are conflated.

During the research work associated with this thesis, a number of other publications were made jointly by the author and other members of the empirical modelling group. These are [BC95, GYC$^+$96, BC97, BCCY97, ABCY98].

Chapter 2

Empirical Modelling Principles and Geometry

2.1 Introduction

This chapter presents the conceptual basis and background to the work in the rest of the thesis. Geometric modelling on computer systems is commonly used in applications that support engineering and architectural computer aided design, computer games and computer generated animations. These are all well-covered fields of research with a significant volume of associated literature. The principal aim of this thesis is to find a way of uniting some of these well-developed research areas with empirical modelling principles. In comparison with tools developed with procedural and declarative programming techniques, many case studies [BBY92, NBY94, BJ94, BSY95] have shown that uniting these principles with various application areas leads to tools for modelling and simulation that exhibit an unprecedented degree of interactivity.

The background to this thesis can be classified into two distinct areas. These areas are discussed independently in Section 2.2 and are briefly described below:

1. Relevant literature on geometric modelling and computer-aided design. Particular attention is paid to notations for geometric design that allow for the textual description of geometric shapes.

2. History and background to the Empirical Modelling Project and its relevance to geometric modelling.

The empirical modelling process on a computer system benefits from strong visual metaphors for the representation of models, to aid a modeller's interpretation of the current state of a model and to assist their conceptualisation of model behaviour through interaction with a model. The construction of computer-based *cognitive artefacts* [BC95] with empirical modelling principles is introduced in Section 2.3. These artefacts are contrived to imitate interaction with a real world referent. The process of animating artefacts benefits from strong geometrical representations, as does support for the collaboration agents by providing many different views and interfaces for interaction with an artefact. In Section 2.3.2, a digital watch and other timepieces are used as the basis of a case-study to demonstrate the construction of cognitive artefacts using empirical modelling principles.

The process of constructing geometric models, as well as the process of describing and reasoning about abstractions associated with geometric models, can be supported by applying empirical modelling principles and definitive programming techniques. Many dependencies arise as constraints between the components of geometric entities, such as:

- two lines have a common end-point;
- two lines are set at perpendicular angles to one another;
- sides of a regular polygon all have the same length.

Section 2.4 examines the process of modelling abstract geometry on a computer system from an empirical modelling perspective, as well as the realisation of this geometry by lines and faces, vertices and point sets. In Section 2.4.1, consideration is given to the use of abstraction by designers in the design process. The CADNO definitive notation for the representation of geometry is introduced in Section 2.4.2. A case-study demonstrates the use of the CADNO notation to represent the abstract geometric relationships in a model of a table[1].

In the final section of this chapter (2.4.4), a brief description of the *EdenCAD* tool [Car94a] is given. This tool, implemented to work as a modular part of the *AutoCAD* computer-aided design package from AutoDesk [Aut92a] allows a user to establish dependencies between defining parameters for geometry from within the package and offers support for empirical modelling. The relationship between EdenCAD and the work in this thesis is discussed[2].

2.2 Background to Empirical Modelling with and for Geometry

Geometric modelling on a computer system is a well-established area of research that has developed alongside advances in the capabilities of computer hardware, particularly computer-graphics hardware. Many geometric modelling software tools are available for Computer-Aided Design (CAD), Computer-Aided Manufacturing (CAM) and Computer-Aided Engineering (CAE). They are often expensive and sold into a competitive market, usually on the strength of their graphical user-interface. Empirical modelling, in contrast, can be viewed as a novel approach to programming a computer system that is established as a research project located at one university.

[1] The kind of table used here is a dining table or a desk, rather than that used to tabulate data or organise information in a thesis.

[2] It is coincidence that EdenCAD is developed by my namesake Alan Cartwright.

As a result, there is a wealth of literature relating to CAD/CAM/CAE and only a relatively small local group of papers relating to empirical modelling.

This thesis discusses issues related to bridging the gap between geometric and empirical modelling through the cross propagation of concepts. This process essentially involves improving data representations and efficiency of the implementation of empirical modelling methods, to bring them into line with geometric modelling methodologies. It is not possible to approach this work from the opposite perspective, by adapting existing CAD/CAM/CAE tools for empirical modelling, as these tools exploit aspects of conventional programming paradigms that conflict with empirical modelling priciples. The approach adopted here is to explore the scope for extending empirical modelling to encompass abstractions in geometry.

A brief background and introduction to current CAD techniques and tools is presented in Section 2.2.1. The two main data representation techniques (B-rep. and CSG) for geometric modelling are described, along with recent work on implicit shape representations. Empirical modelling involves the identification and expression of observed dependencies between data. In existing geometric modelling tools, techniques exist to support similar expressions of observed dependency and these are known as *parametric* and *variational* modelling. In Section 2.2.2, these techniques are distinguished from the representation of dependency in a definitive script.

Section 2.2.3 describes the history of the Empirical modelling project. The work described is based at the University of Warwick and the associated references are, therefore, mainly local to Warwick. Although there are significant similarities between spreadsheets and programming with definitive notations, definitive programming methods form a paradigm for programming whereas spreadsheets are applications written with more conventional procedural programming techniques. The types of data stored in spreadsheet cells are generally simple types such as integers, floating point numbers, dates and strings. The definitive programming paradigm has the potential to support more complex data types[3].

2.2.1 Geometric Modelling

The development of geometric modelling on computer systems has developed alongside the development of computer graphics hardware. One of the earliest tools for two dimensional drafting on computer systems is Sutherland's *Sketchpad* [Sut63], the first proposal for a human/computer interface through graphics. Manufacturing industry developed many Numerical Control (NC) systems for controlling manufacturing equipment, such as lathes and milling machines, by computer program. This prompted much commercial work on *sculptured surface modelling*. This included Bézier's work for Renault on the development of *Bézier Surfaces* [Far90]. At the time, there was a need to be able to simulate the NC process on computer systems prior to manufacture because of the potential for errors in the NC code. This initiated study into how to model and represent three-dimensional geometry of apparently solid material on computer systems.

The addition of the third dimension proved to be a challenging problem. During the 1970s, there were two distinct research groups investigating the representation of three-dimensional geometric shape on computer systems. The product of this work is two separate representations:

CSG *Constructive Solid Geometry* techniques represent shape on computer systems as a finite number of boolean set operations (union, intersection, set difference) on half spaces defined by algebraic inequalities. Based on work carried out at the University of Rochester by Voelcker and Requicha, the seminal journal paper on CSG techniques is by Requicha [Req80]. Similar half-space represen-

[3]This is demonstrated in the ARCA, DoNaLD and SCOUT implementations, described in Section 2.2.3.

tations were developed independently by Okino et al [ONK73].

B-Rep *Boundary Representations* of shape consisting of facets that are subsets of planar, quadric or toroidal surfaces were developed at the University of Cambridge by Braid [Bra79].

Requicha et al developed the PADL-1 notation [VRH$^+$78] (the *Part and Assembly Description Language*), implemented as a computer based tool for constructing CSG models. Shape is constructed by textual description, where the construction sequence is represented by a sequence of assignments to variables in the model. These variables can be associated with defining parameters of the model and its component shapes. PADL-1 is mainly regarded as a research tool that does not support the full coverage of geometry required in industry[4]. This has led to the development of PADL-2 notation [Bro82] and its prototype implementation P2/MM, a computer-based tool for solid modelling with support for both CSG and boundary representation models. PADL-2 supports data types for primitive solid shapes (block, wedge, cylinder, sphere, cone), primitive faces (plate, disk, cylinder face, sphere face, cone face), half spaces defined by surfaces (planar, cylindrical, spherical, conical) and line segments.

The PADL-2 notation was widely adopted and developed by many industrial companies during the 1980s, as noted by Sheridan in [She87]. McDonnell Douglas and other industrial partners devoted a large amount of effort to improving the rather academic and text-based implementation to create an application with graphical user interface support for the modelling process. This application allows a non-programmer to select points by choosing them on the screen representation of the model and click on icons to select shape primitives and operations. This removes the need to write code in order to construct models.

[4] See "*A tale of technology transfer*" by Voelcker, an inset to [Bro82].

Cadetron developed *The Engineering Works* [She87], the first solid modelling package to run on the PC. The package incorporates PADL-2 as its kernel modeller and data representation. This implementation dispenses with the textual construction of models, in favour of an interactive graphical interface with multiple windows.

The data representations for PADL-2 only allow geometric data and some standard attributes (colour, bounding box) to be associated with geometric entities. Geometric data may be appropriate for the design of shape, but for other processes such as the generation of NC machining code and finite element material stress analysis, information relating the features in the geometry with its components is also required. For example, a machine tool may be able to machine edges accurately to a certain tolerance but can drill holes with much greater accuracy. It is therefore necessary to know which edges in the geometric model relate to drilled holes and which edges to the boundaries of the object. A data representation technique for feature based modelling is described by Ansaldi et al in [AFF85, AF88].

The development of graphical user-interfaces has significant benefits for mechanical engineers who are not expert programmers. However, it is not possible to program with a graphical interface for the purpose of integrating solid modelling within other applications. Bowyer developed the sVLIs [Bow94] geometric modelling kernel modeller as a C++ library to support integration of CSG modelling into bespoke computer-based modelling tools. The *ACIS* kernel modeller and programming toolkit from Spatial Technology is widely used for B-rep modellers [Cor97]. Bowyer et al also developed *Djinn* [BCJ$^+$95, BCJ$^+$97], an *application programming interface* (API) to standardise procedure calls to libraries of procedures where the representation is independent of language or point sets, regardless of the underlying software or hardware. Paoluzzi et al [PPV95] have carried out research into the use of functional programming techniques in the construction of geometry. Programming techniques and interfaces to solid modelling allow a programmer to construct solid geometric models, build new application domain specific user interfaces for geometric modelling and perform simulations or generate animations from solid geometric models. The *CAS.CADE* C++ programming library[5] from Matra Datavision allows a programmer to construct bespoke modelling tools and is also used by Matra Datavision as the underlying library for their latest family of modelling software products.

Sculptured surface modelling techniques (Bézier surfaces, B-spline surfaces) can be merged with solid modelling techniques using boundary representation. It is not yet possible to completely integrate these techniques completely with CSG modelling and there is a divide between CSG solid and B-rep surface modellers. This has been partially achieved by Kirshnan and Manocha [KM96] (by plugging NURBS bounded solids into the CSG-tree) and by Berchtold and Bowyer [BB98] (by integrating support for Bezier surfaces and CSG modelling). It is interesting to note Shah and Mäntaylä's comment on the future of modelling programs and the separation of CSG modelling and sculptured surfaces [SM95]:

> "More lately, *implicitalization techniques* have been introduced that may eventually make it possible to merge sculptured surfaces also in CSG models."

Implicit techniques are adopted in this thesis, and implicit and CSG techniques are merged in the case-studies in Chapters 7 and Chapter 8. Implicit surface representation techniques, the *function representation* of shape, are a current topic for research by Pasko, Savchenko and their colleagues [PSA93, SP94, PASS95, MPS96]. In general, point sets constructed from any representation of shape in Euclidean space can be combined into this uniform representation of shape that includes all the standard CSG operations and many more. Traditional CSG mod-

[5] See `http://www.matra-datavision.com/`.

els, closed boundary representations, volumetric objects [KCY93] and skeletal-based implicits [BBCG+97] (also known as *blob-tree* models [WvO97]), in common use in computer generated animations, can all be combined with the function representation of geometric shape. The rendering of shapes represented by function representation require the arbitrary and repeated sampling of mathematical functions at points in Euclidean space, a procedure that is difficult to optimise.

The definitive notation "*HyperJazz*", developed by Adzhiev et al [APS96], is of particular interest in this context. This notation allows a modeller to design shapes using function representation and express dependencies between the defining parameters of the shapes. The notation requires its users to have a good understanding of the function representation of point sets and it would be advantageous if tool included a library of geometric primitives that could be drawn upon during modelling. It would also be beneficial if it were possible to express dependencies between shapes as well as between defining parameters. The latest version of HyperJazz, entitled "*HyperFun*", contains better support for geometry but no longer supports the expression of dependency between parameters.

2.2.2 Variational and Parametric Modelling

The expression of dependency between defining parameters and geometric entities is supported by many geometric modelling tools. Mathematical expressions of desired relationships between numerical variables express the constraints between component entities of a model. These include:

- constraining two points so that they are the same distance apart;
- constraining two lines to be parallel to one another in a particular plane;
- constraining the radii of two circles to be proportional to one another (circle A has twice the radius of circle B).

The terms *parametric model* and *variational model* are used almost interchangeably to describe models containing geometric constraints. Their differences are explained later in this section.

The overall design process for parametric and variational systems is similar. It is described by the following four stages[6]:

1. A user creates the nominal topology of a design using standard geometric and solid modelling techniques. The result is a model with the same underlying combinatorial structure as the final geometry, but without exact dimensioning. In other words, all component entities are connected in the way that they will be connected in the final model.

2. The user describes the relationships between entities in terms of geometric constraints.

3. Once the constraints are specified, the modelling system applies a general constraint satisfaction procedure to produce an evaluated model with dimensions. The system may be over-constrained leading to no solution, or under-constrained leading to many possible evaluations.

4. The user can create variants of the model by changing the values of constrained values, or even the constraints themselves. Each change creates a new evaluation of the model through the re-execution of the complete constraint solving procedure of stage 3.

It can be seen from the stages above that this process is very different from propagation of change in a spreadsheet, where only values in cells that require re-evaluation are computed. If every change in a spreadsheet required the re-evaluation

[6]Process as described by Shah and Mäntaylä [SM95].

of every cell, the system would be seen as inefficient. The definitive programming process is closer to the construction and use of a spreadsheet model than the parametric/variational design process, yet it can represent dependencies similar to those expressed through geometric constraints using definitions[7]. Moreover, with the definitive approach it is possible to express relationships between the topology of the models during the conceptual design phase, integrating stages 1 and 2. This integration process is discussed further in Section 2.4.

PADL-1 [VRH+78] implements a constraint satisfaction mechanism in a procedural way. A sequence of assignments to variables of a model constructs the model. Changing one of the values in the construction sequences and re-executing this sequence results in a new evaluation of the shape. The order in a procedural script of definitions is important and the flow of computation describes models that are *unidirectionally parametric*. The description of the topology of the geometry has to be expressed with references to its defining parameters, leading to the rigid coupling of stages 1 and 2 of the design process.

Flexible constraint satisfaction is independent of the construction order of the geometric model. Parametric systems apply sequential assignments to variables in a model, where every assignment is computed as some function of previous assignments. The ordering of the sequence of assignments is determined by an algorithm. For example, to constrain a point (x, y) in two dimensional space to lie on the line passing through the origin $(0, 0)$ and point $(2, 1)$, the mathematical constraint $y = \frac{1}{2}x$ is used. This system works well unless the value of x depends on the value of y in another constraint. One solution algorithms is the graph method devised by Serrano [Ser87]. Another, called *DeltaBlue*, is credited to Freeman-Benson et al [FBMB90].

[7]The expression of some constraints may require the use of *Higher-order dependency* as described in Section 3.2.2 of Chapter 3.

Variational modelling systems represent geometric constraints using sets of equations. These equations are solved simultaneously to evaluate the dimensions for a model. To constrain a point (x, y) as above, the equation $x - 2y = 0$ is solved by numerical or symbolic methods along with the equations for all the other constraints. One possible numerical algorithm is given by Press et al [PFTV92] — this is based on the popular *Newton-Raphson* solution method. The algorithm iterates through a series of estimated solutions until the estimation is accurate within a certain tolerance level. This algorithm requires an initial guess for the solution. Another approach is to use symbolic manipulation to solve the system of equations, as described by Kondo [Kon92]. In both the symbolic and numerical cases, it is required that there are mechanisms to deal with sets of equations that are over- or under-constrained.

Whether variational or parametric modelling techniques are implemented in a computer-aided design system, the system solves a set of constraints with an algorithm and is conceived in the framework of conventional procedural or declarative programming paradigms. For instance, the GEOMAP-III parametric modelling tool [KSW86] allows a user to represent constraints between geometric entities using a logic programming notation, solving the constraints using inference rules. In contrast, the definitive programming paradigm, founded on empirical modelling principles, represents dependencies as observed from an agent viewpoint and not as logical propositions. This representation encourages open development at the conceptual design phase without the need for early circumscription to topological or mathematical relationships. Stage 3 of the parametric/variational design process imposes a barrier to open development resembling the separation established by compilation of source code to executable binary. Compilation is not required in the definitive programming paradigm.

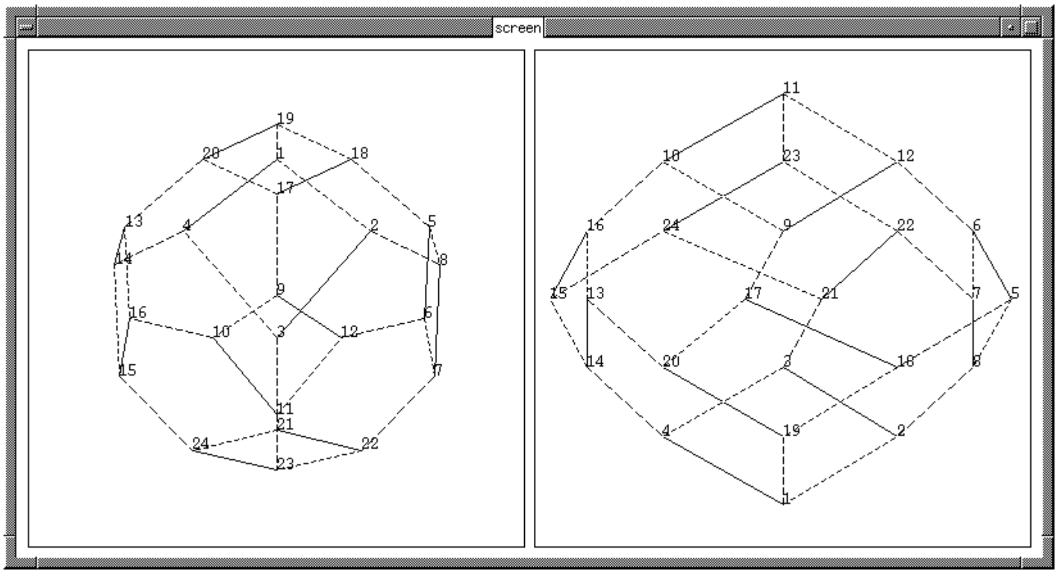


Figure 2.1: Three dimensional Cayley Diagrams generated by the ARCA tool.

2.2.3 The Empirical Modelling Project

The Empirical Modelling Project (EMP) at the University of Warwick originated from a study by Beynon in 1983 of a tool interactively to display and manipulate Cayley diagrams [Cox65], entitled ARCA [Bey83, Bey86a]. Two Cayley diagrams rendered by this tool are shown in Figure 2.1. The research report describing ARCA contains a detailed description of a context free grammar for a notation that supports both definition and declaration of variables. These variables can be of type integer, vertex, colour and diagram. This notation is of interest in the context of this thesis as it includes a *moding* mechanism for handling the dependency between components of data structures in the underlying algebra of the notation.

Interest in the general concept of definitive notations by Beynon resulted in the development of EDEN (the **E**valuator for **DE**finitive **N**otations), a computer program that is an interactive evaluator for generic definitive notations [Yun90] (see Section 1.2.1). This tool is a script interpreter which in use is similar to interaction with a shell program on a UNIX system. Users of this interpreter can at any point in their interaction:

- create a variable or modify the value of an existing variable;
- create a dependency between variables by introducing a new definition or modifying an existing definition;
- create code for a function or modify a function that is or will be used as part of a definition;
- create or modify code for a procedural triggered action;
- inspect the current values of variables, their current definition and the current code for functions and triggered actions.

The syntax of EDEN has control constructs that mimic those in the ANSI C programming language, with *for*, *if-then-else* and *while* statements. Triggered actions are necessary to reveal the current state of the value of definitive variables in the script. When the value of a variable is updated, an action can be set to be triggered as a result of this update. By analogy with a spreadsheet, actions perform an equivalent operation with the updating of one spreadsheet cell causing the display of other cells dependent on it to be redrawn, once they have had their value recalculated. Triggered actions are required as an interface between a definitive script in the definitive programming paradigm and a procedural operating system based in a traditional procedural paradigm, through textual and graphical input and output.

The design of a definitive notation for interactive graphics called DoNaLD (the **D**efinitive **N**otation for **L**ine **D**rawing) is described in [ABH86]. The data types of the underlying algebra of DoNaLD are geometric entities such as points, lines and circles. Dependencies between these entities can be expressed using definitions, as illustrated in Figure 2.2[8]. An implementation of the notation through translation into EDEN definitions is described by Beynon and Edward Yung in [BY88b]. Communication between the translator and the EDEN interpreter in this implementation is via the UNIX command pipeline. For many years, this translation methodology remained a mechanism that could be used to implement newly designed definitive notations or improved versions of existing ones. For example, there is a new implementation of the ARCA notation with translation into EDEN on-the-fly [Bir91] that uses the drawing procedures of DoNaLD.

The SCOUT notation (notation for **SC**reen Lay**OUT**) for screen layout by Simon Yung [Yun93] is another definitive notation that is implemented using a similar translation mechanism. On initialisation, SCOUT opens a window within the screen of the current window-manager. It then becomes its own definitive window-manager within this space. SCOUT can control the layout of sub-windows, buttons, text and provide a *viewport* for DoNaLD drawings. Viewports provide zoom and pan facilities for DoNaLD 2D drawings. SCOUT has no built-in capability to draw shapes and lines.

Issues in the design of DoNaLD and its use as an interactive graphics language were evaluated in [Bey89], a report that addresses the problems of creating and referencing a recursively described shape, such as an infinite staircase. Apart from the first stair, each stair of the staircase is defined to be the same as the one below it, translated by a suitable vector. The report poses an open question: How is it possible to make reference to the n-th step of the staircase?[9] The implementation method for DoNaLD cannot support recursive shape description or reference.

[8] In this screen snapshot figure, the DoNaLD definitions are shown in the window labelled "`Tkeden: DoNaLD Definitions`". The section of code shown is the definition of the flag on the chess clock's face that is pushed up by the minute hand and then drops the instant that the minute hand reaches the vertical position.

[9] For example, how can I hang a bucket on the 56th step, or animate a ball rolling down the stairs from the 109th step?

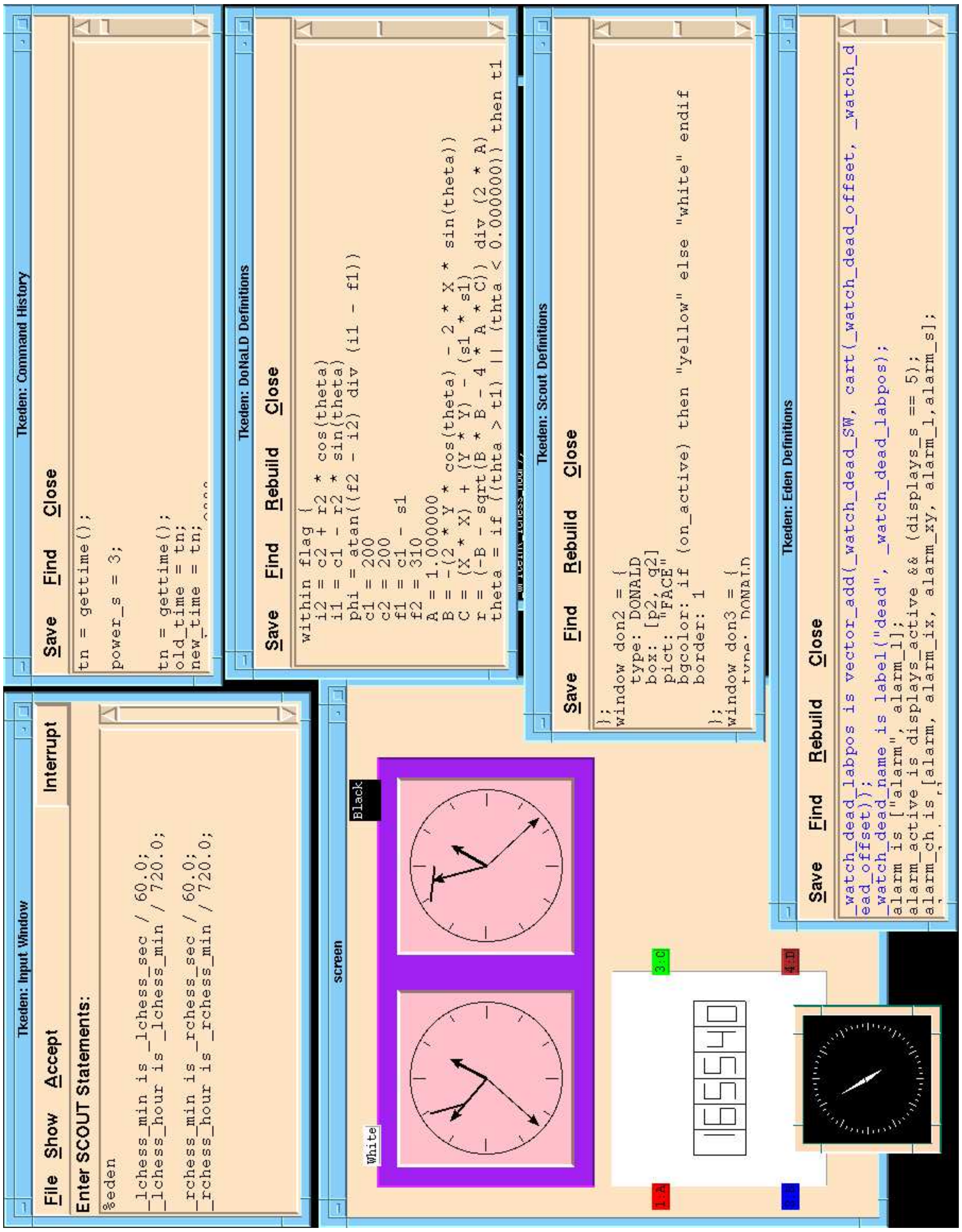


Figure 2.2: Chess clocks constructed from the digital watch model.

Ideally, DoNaLD should also incorporate definitions for shape description like those used by van Emmerick et al in their hypertext approach to the interactive design of solid models [vERR93].

DoNaLD has potential to be used in the engineering design process as a tool for drafting and displaying models, where the dependency between geometric elements can be linked to observed dependency in the model's real world referent[10]. This resulted in research by Beynon and Alan Cartwright into new definitive notations which were specifically aimed at CAD, rather than just simple line drawing tools [BC89]. The CADNO definitive notation (**C**omputer-**A**ided **D**esign **NO**tation) for the description of combinatorial structure and coordinate geometry is described in [BC88] and in Section 2.4.2. A partial implementation of the notation was made by Stidwill for his third year project [Sti89]. This implementation is based on translating CADNO definitions to EDEN definitions. The tool produces three dimensional, wire-frame, black and white graphical output of the geometry represented by scripts of CADNO definitions.

More recent research into the use of definitive notations for CAD software examines the design of definitive notations that support interactive conceptual design [BC92] and concurrent engineering design [BACY94a, BACY94b]. Redefinitions in a definitive script can represent the actions of different agents. This means that the privileges for redefinitions can represent the roles of different designers in the design process. The LSD specification language [Bey86b] is used to describe agent privileges for redefinition in a script. This process is called *agent-oriented modelling.* An agent has definitions in a script that correspond to its current state, definitions of the states of other agents that in can observe, its privilege to change other agents definitions and protocols for action. LSD specifications can be animated through the use of the Abstract Definitive Machine (ADM) [Sla90]. The roles of different

[10]For example, if a lamp is resting on a table, then moving the table will also move the lamp.

designers in the design process can be specified using LSD. Adzhiev examines the roles of different agents in the implementation of CAD systems in [Adz94].

Collaboration between Beynon and Adzhiev initiated further research into definitive notations for interactive shape modelling using implicit function representation techniques. The *Hypersurf* tool [PSA93] developed by Adzhiev and his colleagues provides an environment for experimenting with implicit function representations of shapes. The textual description of shape in Hypersurf has many similarities with a definitive script: the association of a left-hand-side identifier for a function representation with its definition. This motivated further discussion of the benefits of definitive programming techniques for the representation of geometry using function representation.

In my undergraduate project work, I designed and implemented a new definitive notation entitled *CADNORT* [Car94b]. This notation is based on a combination components of the syntax of CADNO and *Hypersurf*. The implementation supports textual interactivity with a notation for the design and encapsulation of the function representation of shape on-the-fly. It is possible to create libraries of shape primitives. Only limited graphical output of these models is available in the form of visualisations of two-dimensional slices through point sets, a task for which the EDEN interpreter is particularly inefficient. CADNORT raises many issues concerning the use of complex data types in definitive notations. These are discussed in Chapter 3.

The line of development of the tools SCOUT, DoNaLD and EDEN has culminated in an amalgamated tool entitled *Tkeden*[11]. This tool integrates the SCOUT, DoNaLD and EDEN front-ends with the EDEN engine for definition maintenance. It offers an attractive window interface environment for the interactive management

[11] The **Tk** in *Tkeden* refers to a switch in implementation strategy made by programmer Y. P. Yung from basing EDEN on the X Windows programming libraries to using Ousterhout's *Tk/Tcl* library. See [Ous90, Ous91].

of definitive scripts, as well as being less demanding on system resources than previous versions of EDEN. Figure 2.2, to be discussed in detail in Section 2.3.1, shows the *Tkeden* tool in use. The current implementation of the graphical tool does not support the UNIX shell pipeline. The result of this is that writing new definitive notations becomes a complex software development task, requiring a developer to have an in-depth understanding of the *Tkeden* source code.

The term *empirical modelling* was adopted for the work in 1995. This change was intended to make a clear distinction between conventional mathematical modelling and the phenomenological, experiential models described by definitive scripts. The adoption of the term reflects the fact that there are some primitive underlying concepts that bring coherence to the diverse activities of modelling using definitions, which has a deeper purpose in trying to model the world. These primitive underlying concepts are common to *Definitive Programming*, *Observation-Oriented Modelling*[12] and *Agent-Oriented Modelling*.

2.3 Cognitive Artefacts

A *cognitive artefact* is an object or an environment that is contrived to represent (through metaphor) interaction with some other (typically real-world) system[13]. An artefact exhibits different states, each of which corresponds to a state of the system that it represents. Interaction with an artefact in this mode of representation allows a human interpreter to explore the correspondence between the artefact and its referent beyond preconceived limits. This interaction is used as a well-established means of communication in many disciplines and is distinguished from communication via documents expressed in a formal language. Examples of artefacts include

[12]For an explanation of *Observation-Oriented*, see `http://www.dcs.warwick.ac.uk/modelling/hi/principles/observation.html`.

[13]Definition for *Cognitive Artefacts* given in [BC95].

models of engineering systems, architectural models, molecular models, maps and globes.

Cognitive artefacts serve a useful function in the general design process (rather than the geometric design process in particular) in two different ways, listed below. Both kinds of representation are especially significant in relation to modern trends towards concurrent engineering [BACY94a, CB91] and several different artefacts are necessary to take account of a whole range of design participants, users and scenarios. The two purposes of artefacts in the design process are:

- artefacts with which the designer can interact in order to develop an insight into the nature and current status of an object being designed;
- artefacts that inform the designer about the current status and progress of the design process.

The purpose of discussing cognitive artefacts in this thesis is to explore the empirical modelling process and its requirement for good metaphors for interaction that can be provided through multi-media interfaces. Solid geometric models can provide interaction with computer-based artefacts that closely imitates interaction with the real world referent. Current tools that support the empirical modelling process provide only simple line-drawn graphical models. Through understanding the relationship that exists between artefacts and their graphical representation, there is potential to support empirical modelling for richer interaction with and more detailed construction of artefacts. This process will benefit from better geometric models to assist in the communication of the model. The relationship between empirical modelling and cognitive artefacts is explored in Section 2.3.1 and a case-study examining cognitive artefacts for timepieces is presented in Section 2.3.2.

Many different agent views of the same artefact can exist. For example, in the mechanical design process a component part may be the responsibility of a geometric designer, an electrical engineer, a materials expert and others. Each agent has its own viewpoint of the artefact and the nature of their interaction with the artefact is dependent on their viewpoint. Issues in the use of artefacts as a means of communication between agents are discussed in Section 2.3.3.

2.3.1 Cognitive Artefacts and Empirical Modelling

Empirical modelling can be used for constructing artefacts by specifying each state of a referent in terms of the current values of a collection of observables[14]. Primitive interactions with the referent by manipulation of the observables are specified as synchronised changes to these observables. The artefact has its own collection of observables and primitive interactions that correspond with the observables and primitive interactions of the referent. This presumes a metaphor for the representation of the current state of the referent, whereby observables (variables) of the computer model correspond to observables of the referent. The use of such metaphors can be illustrated by considering different representations of time. For example, consider the following three metaphors for time:

- the position of hands on a clock;
- the value on a numerical display;
- the position of the sun in the sky inside some virtual environment, such as a computer game.

The exact and explicit correspondence between states and transitions in artefact and referent plays a fundamental role in the empirical modelling method, and is the basis for being able to recognise the association between them. It is through experimentation with the artefact that the characteristic relationships between ob-

[14]This content of this section is based on work previously presented in [BC95].

servables that determine the identity of the referent are revealed. Traditional programming techniques render the current state of the screen that represents a model using procedural algorithms. The empirical modelling approach is to treat the state of the screen as part of the computational state of a model. The state is defined by a system of variables, whose values are appropriately synchronised in change to represent the dependencies between graphical elements. These graphical elements are often of a geometric nature, such as lines, circles and polygons.

It is the representation of dependency, both between graphical elements and the observables that form computer-based artefacts, and the linking of these two kinds of dependency, that facilitates construction and animation with artefacts. Through the use of this dependency, it is possible to represent the effect of spontaneous human agent interaction in a system, continuous processes and instantaneous events. The geometric primitives available in current empirical modelling tools such as DoNaLD do not allow for the representation of geometric shape in more than two dimensions by techniques such as boundary representation and CSG methods[15], or for adequate representation of dependencies between components of shapes. As a result, current support of metaphor for the representation of artefacts using empirical modelling on a computer system is limited, and not generally suitable for tasks such as mechanical design. These tasks may require three-dimensional models of shape to achieve the goal for the construction of artefacts: to place into correspondence the states and state transitions of the artefact and referent, and to be able to recognise the association of the artefact with the referent through interaction with the artefact.

[15]See Section 2.2.1.

2.3.2 Case-Study - *Timepiece Artefacts*

Figure 2.3 depicts three artefacts representing aspects of timepieces: a digital watch, an analogue clock and a statechart (cf. Harel [Har87b, Har87a]), which represents the functionality of the digital display. The clock faces are artefacts for communicating time and the statechart serves to tell a designer how the display functions of the digital watch are affected by the pressing of watch buttons and the level of charge of the battery. In Figure 2.3a, the digital watch face is surrounded by four buttons that respond to mouse button clicks. The buttons are linked by definition to the statechart in Figure 2.3c, where the current internal state of the watch is given by boxes with solid lines rather than dotted lines.

In a real analogue clock, the hands of the clock may be coupled mechanically so that when the hour hand moves, the minute hand moves with it. This dependency can be expressed with a definition for *angle_hour_hand*[16], the angle between vertical and the hour hand measured clockwise, in the following way:

$$angle\_hour\_hand = \frac{angle\_min\_hand}{2\pi} \times \frac{\pi}{6} \tag{2.1}$$

This dependency is represented in the analogue clock face in Figure 2.3b in a graphical line drawing representation, where the angle of the second hand *angle_sec_hand* is linked to the angle of the minute hand in the same way. The time on the digital watch shown in 2.3a is linked by a definition of *angle_sec_hand* to the number of seconds through a day. This is calculated from the time display of the digital watch.

Sundials are timepiece instruments that could be represented with the support of three-dimensional graphical representations of artefacts. Time is represented on a sundial by the shadow of a pointer cast by the sun on to a graduated disc. The pointer is raised above the flat disc. An environment could be constructed for the

[16]The angle of the minute hand *angle_min_hand* is measured modulo 24π.

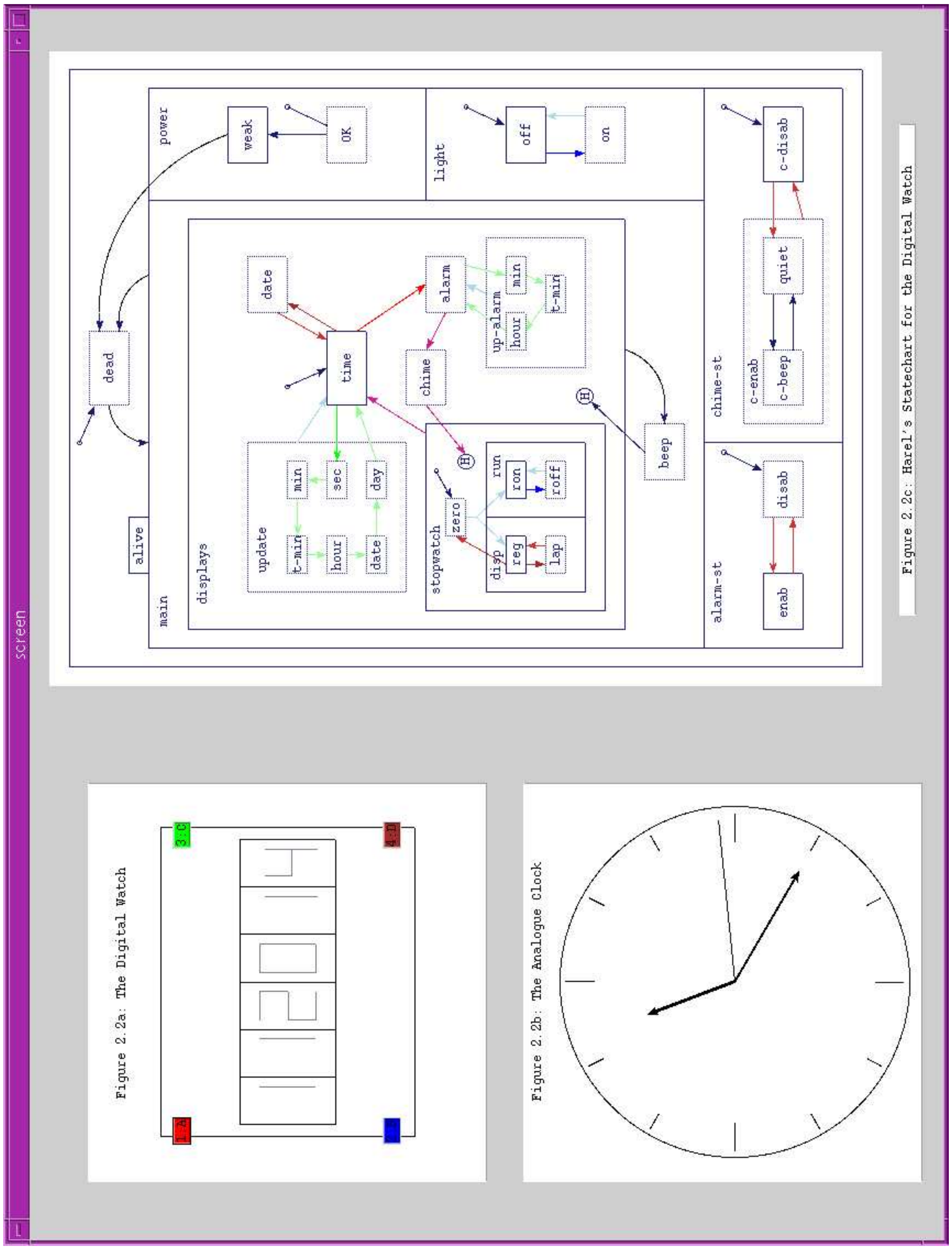


Figure 2.3: View of the *Tkeden* cognitive artefact for a digital watch, analogue clock and statechart.

exploration of sundial artefacts, where the rotation of the earth causes the apparent motion of the sun throughout the day. In this environment, it is possible to explore agent views of time by:

- correctly and incorrectly positioning and orienting sundials;
- the effect of breaking or modifying the geometry of the pointer of a sundial;
- examining an agent's misconceptions of time when glancing at the graduations on the disc from an oblique angle;
- comparing the margins for error in viewing time during different seasons, in bad weather and in different countries of the world.

Characteristic patterns of change in observables in the referent model are identified through observation and experiment, and are always subject to revision in the light of subsequent observation. The process of developing an artefact naturally involves incremental construction that is guided by evolving knowledge about the properties of the referent. In this process, it is essential to be able to record the current status of partial models, and revise these to take account of new information and perspectives on the referent as they are encountered. Empirical modelling principles can be used to support this process.

Figure 2.2 shows the *Tkeden* development environment in use for the construction of, and interaction with, the model of a chess clock[17]. The figure shows the definition input window (top left), SCOUT definition viewing window (top right), DoNaLD definition viewing widow (middle right), translated to EDEN definition window (bottom right), model viewing window (middle left) and an independent

[17]The nature of development and interaction with the *Tkeden* tool is difficult to convey in a snapshot image. It is only the experience of the tools over time that demonstrates the full potential of the tool. Note that only the interpretation and maintenance of definitions has occurred during the construction of the model and no compilation has taken place

computer system clock (inset bottom left). The model was developed on-the-fly directly from the digital and analogue clock models, reusing the components (analogue clock face, relationship between the hands) of the time artefacts to construct the new chess clock cognitive artefact. The existing clock mechanism is linked to the chess clock model to make the clocks tick and runs in real time, simultaneously with the digital watch model. This on-the-fly construction of models in *Tkeden* supports computer-based rapid prototyping methods for the incremental development and reuse of components of artefacts.

With support for good geometric representations, the tools can provide much better virtual environments for interaction and rapid prototyping. Support for solid geometrical models that can be used as visual metaphors for the communication of the current state of a cognitive artefact and can assist the incremental construction of a modeller's evolving insight into a model. For a clock model, a three dimensional model can be used to experiment with the design of the clock casing or the design of the clock hands. It is also possible to consider inventing a completely new spatial method for the representation of time, such as a spinning globe representing the world with the location of dawn and dusk indicated by illuminated and dull areas. This is well beyond the scope for data representation in the current *Tkeden* tool, which supports only data types for two-dimensional line geometry through translation from DoNaLD definitions to EDEN definitions. The data structure used for the representation of DoNaLD data is a general EDEN data structure that is not specifically designed for the representation of geometry[18].

2.3.3 Agent Views of Cognitive Artefacts

The timepiece models demonstrate how empirical modelling techniques can support three kinds of agent interaction with a model:

[18] Further discussion of the DoNaLD to EDEN translation process is presented in Section 3.4.1.

- Interaction with the model through interfaces provided to represent the interface of the referent. This is illustrated in Figure 2.3a, where virtual buttons that represent the real buttons of a digital watch can be pressed by mouse clicks.

- Revision of the model through open-ended redefinition to represent revision and incremental construction of the model. This is demonstrated by the on-the-fly construction of the chess clocks model, as shown in Figure 2.2.

- Autonomous agents who take action to change the model without the intervention of a user modeller. An autonomous time-updating agent exists in the timepiece examples.

It is also possible to treat the components of a model and their relationship as interacting agents. For example, rather than the *analogue* definition of the relationship between the hour hand and the minute hand expressed in Equation 2.1, we can conceive an autonomous agent that observes the position of the minute hand. Every ten minutes, this agent updates the position of the hour hand.

Harel's vision of software development for reactive systems [Har92] emphasises the role of visual formalisms, in particular the use of statecharts [Har88]. The widespread adoption of statecharts in software development methodologies (cf [Rum91]) is a motivation for interpreting statecharts as cognitive artefacts within the empirical modelling framework. The concepts of state *depth* (a state that can be one of many possible sub-states) and *orthogonality* (a state with several coexisting sub-states) are central to the construction of statecharts. The depth concept can be represented by an agent who plays more and more specific roles in a model, the orthogonality concept by agents cooperating simultaneously. These roles and privileges of agents can be described by the LSD specification notation [Bey86b].

Statecharts as cognitive artefacts can play an important role in the concurrent engineering process [BACY94b, CB91], where several cooperating human agents are collaborating in the construction of an engineering model. The artefact represents the current internal states of the model and allows for the discussion of states and of how the state model can be collaboratively improved by the agents. The statechart can also be used to represent and to compare several prototype models for the same artefact. Visual formalisms assist design agents in the process of bringing together several different versions of a design artefact and committing to components of these to form a new artefact.

Each human agent may have a particular focus on an aspect of the design, e.g. the materials used or the electrical properties of a referent. Practical support for the concurrent engineering process across distributed systems is a current topic for research. No support for associating variables with agents and controlling their privileges to redefine or reference other variables is provided by EDEN. There is further discussion of LSD and support for multi-user modelling in Section 6.3.

2.4 Abstract Geometry from an Empirical Modelling Perspective

Commercial modelling packages place their primary emphasis on geometry and only recently have efforts been made to capture a greater part of human design intent in the construction of models[19]. Feature-based approaches to geometric modelling (cf. [SM95, AF88]) allow the association of high-level information, such as the result of *cutting* a cylinder away from a block of material being a *hole*, with the good mathematical representation of the geometry of the model formed by the boolean set-difference operation for the cut operation. In general, abstracting the

[19]The work presented in this Section (2.4) has previously appeared in a technical research report [BCCY97].

geometry of a real-world object entails discounting some of the characteristics of the real object. It seems that there is a need to liberate a designer from the obligation to specify explicit geometry at the initial, conceptual phase of the modelling process. A designer typically conceives and manipulates an object in terms of its combinatorial features rather than as an abstract point set in Euclidean space.

Geometric modelling can be viewed as the construction of cognitive artefacts for the exploration of shape, both as *metaphor* for communicating structure in a model in the manner of a skeletal sketch, and as a virtual reality object that can be viewed as a *product*. Empirical modelling can be used to support the conceptual and initial phases of the geometric design process, as well as in the specification of mathematical models for geometry. Empirical modelling techniques can also be used to establish dependencies between feature information, combinatorial structure, attributes, design processes and geometric models in an open-ended way. Section 2.4.1 examines the relationship between design process, geometric modelling and empirical modelling. This is followed by the description of the CADNO definitive notation for geometric modelling (Section 2.4.2) and a case-study in the use of the CADNO notation for a *table* (Section 2.4.3).

2.4.1 Meta-Modelling in Design

The concept of *meta-modelling* is introduced by Tomiyama in [Tom89]. He expresses the need for modelling methods that combine the representation of objects with the representation of the design process itself. His vision is a design framework in which there are many descriptions of the design product, all of which are related and draw on a single source of information. To realise this objective requires an intimate integration of geometric and non-geometric modelling in a context that also supports the management aspects of the design process - meeting the need to record versions, alternative viewpoints, archives and libraries. Empirical modelling principles have the potential to represent the dependency between the geometric and non-geometric data and scripts (or sections of scripts) of definitions can be used to record versions, viewpoints and create libraries of parts. These scripts can be managed in the same way that text documents are managed by a word processor or desk top publishing system.

The ADDL system, developed by Veerkemp [Vee92], is one proposal that involves the integration of two kinds of knowledge:

1. Knowledge about objects - their attributes and characteristics.

2. Knowledge about the design process - a designer's intention.

Non-geometric information is recorded in the knowledge base for each object. An important characteristic of Veerkemp's conception is that the designer has a creative role to play in the selection of scenarios that shape the development of the evolution of objects.

The context for the paradigm shift proposed in the work in this thesis is supplied by a conflict between two engineering cultures that are well characterised in the words of Brödner [Brö95]:

> One position, ... the "closed world" paradigm, suggests that all real-world phenomena, the properties and relations of its objects, can ultimately, and at least in principle, be transformed by human cognition into objectified, explicitly stated, propositional knowledge.

> The counterposition, ... the "open development" paradigm ... contests the completeness of this knowledge. In contrast, it assumes the primary existence of practical experience, a body of tacit knowledge grown with a person's acting in the world. This can be transformed into explicit theoretical knowldge under specific circumstances and to a principally limited extent only .... Human interaction with the environment, thus, unfolds a dialectic of form and process through which practical experience is partly formalized and objectified as language, tools or machines (i.e. form) the use of which, in turn, produces new experience (i.e. process) as basis for further observation.

The essential need for interaction between user and computer makes its impossible to construct a computer modelling system that exclusively and faithfully represents a closed-world perspective. This is because experience is necessarily involved in interaction between the user and the systems. A user can interpret system responses in ways that are beyond the scope of what has been formally specified. The openness of interaction between user and computer system can be suppressed by restricting the experimental channels available to the user. The products of the closed world culture are systems that offer preconceived framework for interaction between the designer and the computer. The nature of the content relation that associates the computer model with an external referent is prescribed and the manner in which the computer model is shaped by the designer's interaction is predetermined.

During the modelling process, definitive programming techniques support computer-based modelling that makes explicit provision for both the content relation and the user-computer interaction to be enriched beyond preconceived limits. Is it possible to preconceive which information about an object will be significant in the design process and what exceptional scenarios will need to be considered? In general, the closed world approach to modelling cannot address these and other concerns of meta-modelling. Description of the design of a product from a number of different viewpoints of the design of an object cannot be expected to be coherent and consistent. Interaction with definitive scripts allows for the introduction and testing of new scenarios for designs on-the-fly, with support for "*what-if?*" experimentation, as well as the representation of interdependencies between different viewpoints of a design.

If the profile of a table top is transformed from a square to a circle, at what point does it cease to have corners? Moreover, if reference is made to the corners of the table to connect its legs, what is the new location of the table legs during and following the transformation? Human insight is typically necessary for decision making in design and computer-based tools for design should support this essential involvement of this interpretive activity in the development process. The problems of computational efficiency that arise when dealing with shape as a product have encouraged investigation of closed-world frameworks for shape modelling, where optimisation of algorithms takes priority over situating shape modelling in a wider context.

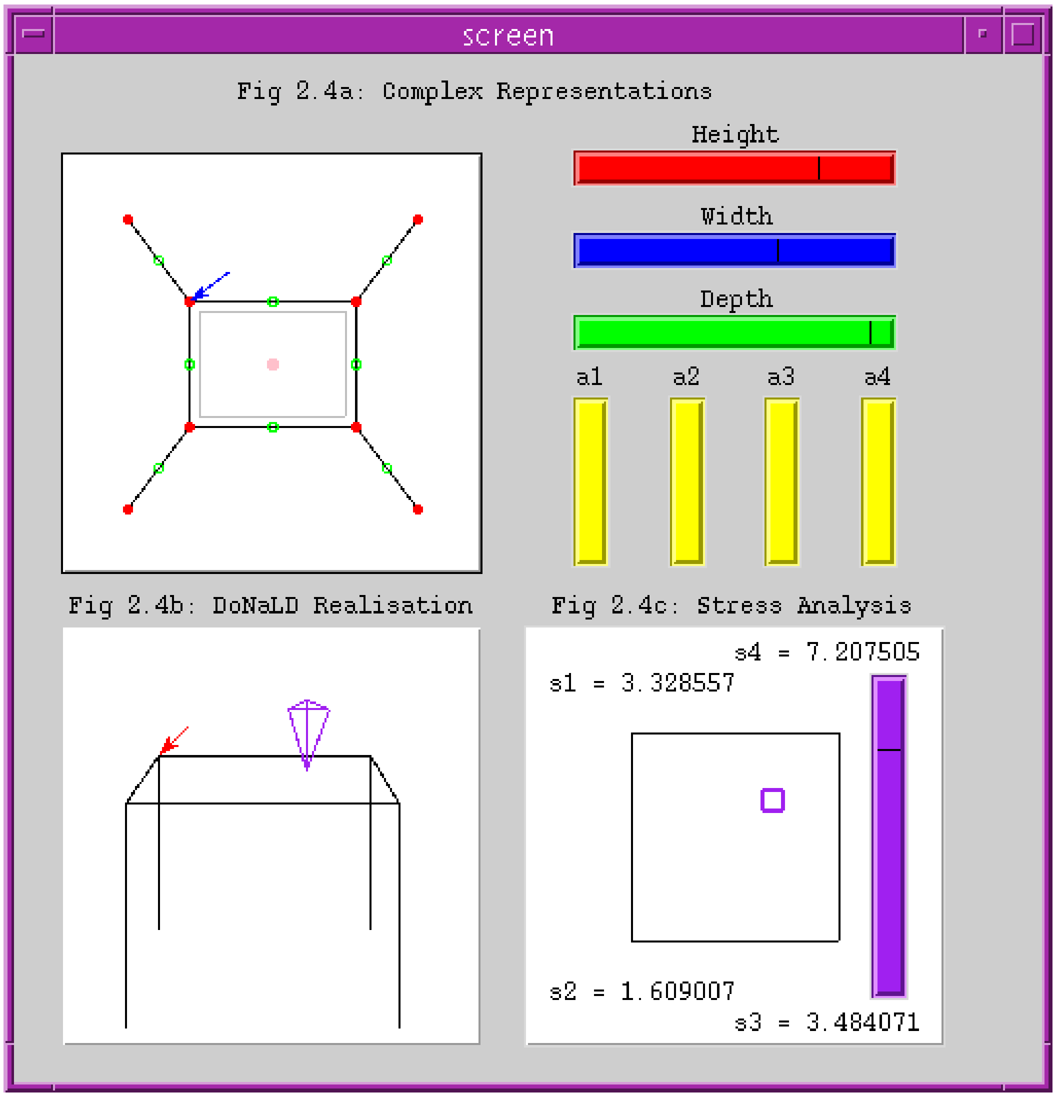
Definitive programming can be thought of as a programming paradigm that provides a spreadsheet-like environment for interaction with computer systems. Chmilar and Wyvill [CW89], and Emmerick, Rappaport and Rossignac [vERR93] demonstrate the potential for applying spreadsheet principles to shape modelling. Nardi's thought-provoking analysis of the spreadsheet paradigm culture [Nar93] indicates that such principles can have an even more radical impact when used outside the conventional programming framework. Conceptual design and design in a concurrent engineering context demand a degree of openness that entails such a shift of programming paradigm. The research in this thesis examines the enabling technologies for and the implications of such a radical paradigm shift to support geometric shape modelling.

2.4.2 The CADNO Notation

The definitive notation for computer aided design (CADNO) introduced in [BC89] can be used to model *abstract geometry*, which places emphasis on the combinatorial and metaphorical aspects of geometry rather than Euclidean point sets. In variational and parametric design, dependencies can be expressed between parameters that define the location of geometry by formulating constraints. These dependencies are determined after commitment to the topological / combinatorial structure of the geometry. The CADNO notation allows for both the expression of dependency between components of combinatorial structures and between scalar parameters. Dependencies can also express the relationship between these two areas and the realisation of geometric objects as graphical representations of Euclidean point sets.

Variables for the representation of combinatorial structure do not necessarily have a realisable counterpart in geometric shape. In an agent view of the DoNaLD notation for line drawing, a designer's mental model of a table object may have corner observables that are end points of lines and each corner can be clearly identified (see Figure 2.4b, to be discussed in Section 2.4.3). These corners are distinguished from points in a wire frame quasi-realistic rendering of a geometric table (as shown in Figure 2.4d) that are observables for the computer application used in rendering algorithms. Paradoxically, there may be no location on a physical table that can properly be called the corner of a table (cf. the table depicted in Figure 2.4d), yet the concept of a corner as a feature of the table is essential for high-level communication about table design.

The data types in the underlying algebra for CADNO are summarised in the table below:



a, b, c

d

Figure 2.4: DoNaLD and VRML realisation of an abstractly defined table.

complex	list-of label $\times$ list-of list-of label
d-complex	complex $\times$ int
frame	list-of vector-of real $\times$ d-complex
object	list-of frame $\times$ (list-of frame $\rightarrow$ (vector-of real $\rightarrow$ boolean))

In general terms, a complex is a combinatorial structure used to specify the abstract geometric entities from which a geometric object is synthesised, such as points, lines, faces and scalar values. A complex can be realised in Euclidean space as a frame, by attaching coordinates and values to its vertices. A geometric object is functionally determined as a point set from a list of frames. A range of different functions can be used for realisation: for instance, a list of frames can be interpreted as the basis of a boundary representation, a CSG model, an implicit function specification, or simply a three-dimensional line drawing[20].

No complete implementation of CADNO yet exists[21] and this thesis does not describe one. CADNO is viewed here as an ideal notation that fits well within the framework of definitive notations conceived prior to CADNO. This thesis examines new ways to approach the implementation of definitive script notations with reference to the technical problems encountered when trying to use existing definitive notations for geometry. This approach leads to the new *empirical worlds* definitive notation for computer-aided geometric design. This notation exhibits many of the same qualities as the CADNO notation. The CADNO notation is compared to the new approach in Chapter 9.

2.4.3 Case-Study - *Table*

To demonstrate the potential for the use of the CADNO notation, a case-study for the modelling of a table is presented. Table 2.1 shows a listing of definitions in

[20] The three-dimensional version of DoNaLD I developed for this case-study was further improved by MacDonald in his third year undergraduate project [Mac97].

[21] An attempt to implement a tool to support the CADNO notation exists. It is written by Stidwill as part of his undergraduate degree project [Sti89].

the CADNO notation that captures the combinatorial structure, point coordinate locations and relationships between scalar parameters of a designer's model of a physical table. This table has four folding cylindrical legs and a square profile table top with rounded corners.

In the following list, some of the definitions in the CADNO script for the model are described:

`table` A three-dimensional combinatorial structure representing eight identifiable reference points `A` to `H` for the table, as well as their interconnections. The legs of the table are represented by the pairs "`[A, E]`", "`[C, G]`" etc. and the table top face as the quadruple "`[E, G, H, F]`".

`angles` The four scalar quantities representing the current angle of fold for each of the four legs.

`table1` An association of coordinate points with the labels in the combinatorial structure "`table`". Notice the dependencies expressed between labels from complexes not yet associated with this frame. These associations occur during the realisation process.

`angle1` A frame that associates a scalar value with the fold angles from complex "`angles`" and represents limits ("`subject to`") for the range of values the scalar values can take.

`pict1` A geometric object on three frames with a realisation as a line drawing in a three dimensional version of DoNaLD. The dots in the braces ("{ .... }") should be replaced with a template description of how to realise the geometry from the frames that the object is based on.

The listing in Table 2.1 is a specification for the geometric models shown in Figure 2.4. The figure is a simple demonstration screen, constructed using *Tkeden*,

```
table = 3-complex on [A, B, C, D, E, F, G, H] with {[A, E], [C, G],
                [D, H], [B, F], [E, G, H, F]}

base = 3-complex on [origin]                      # User's handles
angles = 1-complex on [angle1, angle2, angle3, angle4]

dimensions = 1-complex on [height, width, length]  # Designer's handles
radii = 1-complex on [legRad, cornerRad]

table1 = frame on [table, base]
                where   A = B - height * {sin(angle1), cos(angle1), 0},
                        B = origin + {0, height, 0},
                        C = D - height * {sin(angle2), cos(angle2), 0},
                        D = origin + {0, height, depth},
                        E = F - height * {-sin(angle3), cos(angle3), 0},
                        F = origin + {width, height, depth},
                        G = H - height * {-sin(angle4), cos(angle4), 0},
                        H = origin + {width, height, 0}
                        origin = {0, 0, 0}
radii1 = frame on radii
                where   legRad = 3.0,
                        cornerRad = 10.0
                subject to
                        0.5 <= legRad <= 20.0,
                        legRad <= cornerRad <= 25.0
angle1 = frame on angles
                where angle1 = 0, angle2 = 0, angle3 = 0, angle4 = 0
                subject to
                        0 <= angle1 <= 90,
                        0 <= angle2 <= 90,
                        0 <= angle3 <= 90,
                        0 <= angle4 <= 90
dimensions1 = frame on dimensions
                where height = 40, width = 50, depth = 30
                subject to height > 0, width > 0, depth > 0

pict1 = object on [table1, angle1, dimensions1]
                with extent(donald3d) { .... }
pict2 = object on [table1, angle1, dimensions1, radii1]
                with extent(VRML)
                     {  Cylinder { axis A E radius legRad}
                        Cylinder { axis B F radius legRad}
                        Cylinder { axis C G radius legRad}
                        Cylinder { axis D H radius legRad}
                        Profile { base E F G H extrude 10 }
                        .... }
```

Table 2.1: CADNO script for a table model.

that is contrived to illustrate how CADNO could be integrated with other empirical modelling tools. Figure 2.4a depicts the underlying complex that represents the table: it provides a mode of reference to the three dimensional DoNaLD image in Figure 2.4b of the figure through mouse sensitive spots. The location of a feature of the table in Figure 2.4b, such as the position of the foot of a particular leg, is identified by an arrow when the appropriate spot on the combinatorial structure (`complex`) is selected.

Figure 2.4c illustrates how the geometric models of the table can be used in conjunction with non-geometric models in an integrated manner. It depicts the distribution of weight on the legs of the table (values `s1` ... `s2`) when a load is placed on the table top. The tool supplies an interface through which the load can be relocated by direct manipulation and includes a slider bar to allow its mass to be scaled. The load is also represented by an inverted prism in Figure 2.4b. Other *slider* controls allow for experimentation with the scalar parameters that define the geometry for the table, such as its height, width, depth and the angle by which its legs are folded (`a1` ... `a4`) within the range limits specified in the CADNO script.

A more realistic rendering of the table geometry is shown in Figure 2.4d. The image is generated by first creating a section of VRML code[22] to represent the geometry consistent with its template definition in the CADNO listing, and then rendering this in a VRML browser. Existing dependency maintenance tools do not support good renderings of objects and the most complex graphics currently available in *Tkeden* is three-dimensional line drawing with labelling and attributes (cf Figure 2.4a, b and c).

[22]VRML, the *Virtual Reality Modelling Language*, is discussed further in Chapter 7 and [ANM96].

2.4.4 The EdenCAD Tool

With standard two-dimensional computer graphics hardware, it is impossible to generate a realisation of a high-level parametric description of a geometric object as fast as it is possible to update a line drawing in a DoNaLD image. To improve efficiency, there is a CADNO-like tool developed at the University of Warwick by Alan Cartwright as part of his doctoral research [Car94a], which links a definitive notation with a geometric modeller. *EdenCAD* is a Lisp-based dependency maintainer that makes use of the AutoLISP interface [Aut92b] to the AutoCAD draughting design package [Aut92a]. The idea is that by utilising an existing package to handle the realisation of objects, efficiency in the realisation of geometry can be improved. Dependencies can be established between the defining parameters of the AutoCAD geometry by EdenCAD definitions.

In general, this strategy leads to some loss of control over the representation without the construction of a sophisticated interface to the dependency maintainer. The most common problems are concerned with reference to the components of an artefact. At other times, it is the very intelligence of the software being invoked that poses problems, since it entails an implicit agency that subverts the intended dependency relationships. For example, consider the problems of implementing a definitive notation for screen layout when windows are subject to relocation to respect "intelligent" constraints (such as visible, non-overlapping, automatically scaled).

The work in this thesis takes a different approach to efficient implementation from EdenCAD. It investigates how greater efficiency can be achieved by improving data representations and algorithms for dependency maintenance. Consideration is given to the exploitation of three-dimensional computer graphics hardware, now becoming commonplace in high-end computer workstations and personal computers alike. This hardware can be utilised through software interfaces, such as *OpenGL* programming libraries [WND97].

Chapter 3

Theoretical Issues

3.1 Introduction

In the previous chapter, I have discussed reasons why empirical modelling is good for geometry, and why strong geometry is beneficial for empirical modelling. In this chapter, the technical issues faced in trying to develop the relationships between these two types of modelling are described, in terms of both conceptual issues and the limitations of current tools that support empirical modelling. It is important that the representation of geometric data and dependency between that data in the implementation of such a tool is clearly consistent with the geometry itself. Solving the problems posed by the technical issues is the motivation for the research presented in the chapters that follow.

Interaction with geometric models reveals the integrity of the geometry. It is often the case that what is apparently one piece of geometry is actually an assembly of several component parts. A mechanism is required to manage the components and their dependencies so that interaction, reference and redefinition of the subcomponent parts are consistent with the overall definition of the assembled geometry in a definitive script. The way in which interaction with a geometric assembly affects a copy needs to be appreciated by the modeller. An overview of the issues discussed in this chapter is introduced in the following itemized list.

- The copying of geometric assemblies is a desirable feature in an interactive notation for geometric modelling. What is the status of the internal subcomponent definitions of a copied assembly and its dependency on the original assembly?

- Dynamic instantiation and removal of geometric entities is a desirable feature in some applications, for instance, drawing graph paper with scales that depend on the contents of particular data sets. If component geometry is at one level of definition in a script, are there higher-levels of abstraction in definitions in a script dynamically to control the lower-level geometry consistently with some high-level model?

- Where dependencies exist between geometric entities, these entities can be defined in many different modes. A mode describes the level in the data structure of an entity at which its defining parameters are given by definitions or by construction with constructors. How does this process affect a modeller's interaction with geometry in a modelling environment containing dependency?

- Structure in data can be considered as a form of dependency between values of different types. What is the relationship between dependency and data structure?

- Geometric data is often of a continuous rather than a discrete nature. A straight line can be defined by two end points and it is also a point set defined by a set membership condition of the set of points that lie directly between the end points. A straight line can also have infinite length, where it is parametrised by a vector and a point through which is passes. Dependency between geometric entities may need to be defined in terms of point sets rather than defining parameters. By what methods is it possible to represent geometric *continuous* data in an environment containing dependency on a discrete computer system?

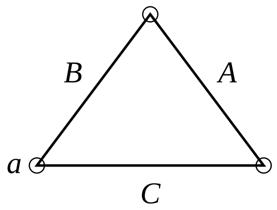
- Geometric data can be defined by parameters, is represented by point sets and is often associated with many attributes for data such as colour, texture, transformation, line width and so on. How is it possible to associate attributes with geometric entities where there is dependency?

Each of these questions is considered in separate sections of this chapter. Firstly, issues relating to empirical modelling tools that capture aspects of geometry are considered Section 3.2. This is followed in Section 3.3 by discussion of the relationship between data structure and dependency, with motivating examples relating to geometry. Secondly, issues relating to integration of good geometry with empirical modelling are considered. Material in Section 3.4 examines issues of data representation, the creation of good computer-based representations for geometric data that can be beneficial to a modeller and consistent with real world experience of geometry. Finally, Section 3.5 presents the basis for a solution to the problems posed by the technical issues raised. This is the basis for the practical work presented in chapters 4, 5, 6, 7 and 8.

Where possible, examples of existing methods for handling the technical problems that arise from the technical issues are presented. Often these methods are only partial solutions and not suitable for the representation of complex geometry. In particular, Section 3.4.1 examines the issues of representing geometric data within the EDEN generic dependency maintainer tool [YY88, Yun90].

3.2 Definitions and Dependency with Geometry

Dependency in geometry cannot be observed through static inspection. It is interaction with artefacts that reveals the integrity of their geometry. A polygon is a geometric entity made up of interdependent lines, where the end point of every line is also the end point of another. This relationship is a form of dependency that is common in geometry. Empirical modelling principles can be applied to represent this dependency in a script of definitions. A polygon can be considered as an *assembly* of integrated subcomponent lines with dependencies between them. An *assembly* is defined as a grouping of definitions in a definitive script that combine to construct an entity that contains subcomponent dependencies. The polygon and its component lines can each be uniquely identified and the indivisible relationships between the lines expressed as definitions in a definitive script.




```
B = line(a, 1 @ pi/3)
A = line(B.2, 1 @ -pi/3)
C = line(A.2, a)
```

Dependency relationships in empirical modelling are acyclic and therefore there is a need to define a starting point on which all the lines depend. Consider the triangle T shown above. With definitions in definitive scripts, it is not possible to define the start of line A to be the end of line B, the start of C to be the end of line A and the start of B to be the end of line C, because this is a cyclic dependency. With the introduction of one more definition for point a it becomes possible to define the start of line A to depend on a and the end of line C to also depend on a. The loop of cyclic dependency is removed in this way.

Triangle T is an assembly that integrates straight line subcomponents. The construction of assemblies may take the form of one definition for the whole assembly, for example "$T = triangle(a, ||A||, ||B||, ||C||)$"[1] or a grouping of separate definitions for each separate line[2]. This section of the chapter examines the issues for the definition and subsequent reference to or redefinition of the subcomponents of an assembly definition.

The first issue discussed in this section (Section 3.2.1) concerns the status of a copy of assembled geometry. In a graphical drawing tool such as *xfig*, when an object is copied its new instance is like a photograph of the original in the state that the original is in at the exact moment of the copy operation. The copy is completely independent of any subsequent changes to the original. If S is a triangle defined by indivisible relationship to be a *copy*[3] of T, i.e. a definition of the form "$S = copy(T)$", the subsequent states of S and T are in some way connected. Dependency introduces ambiguity into the concept of *copy* and a modeller needs to be aware of the subtle qualities of the different kinds of copy possible by definition. Ambiguity can be introduced in respect of three types of activity:

- redefinition of T or its subcomponents;
- referencing subcomponents of T;
- privilege to make redefinitions of subcomponents of S.

The redefinition of T or its subcomponents, making reference to the subcomponents of S and the permission to make redefinitions of the subcomponents of S can all have more than one possible interpretation in terms of the future state of the script and may not match a modeller's expectation of the effect of the copy.

[1] The length of a line x is given by "$||x||$".

[2] The second grouping method is similar to the DoNaLD *openshape* notation construct.

[3] Copying is a general term that admits the possibility of operations such as rotation, scaling and translation of shapes in a geometric modelling environment.

Dependency introduces ambiguity into the concept of *copy* and a modeller needs to be aware of the different kinds of equivalent copy possible by definitions.

In Section 3.2.2, the levels of abstraction for the definition of assemblies are demonstrated. At a low-level of abstraction, a script may have operators and data types for geometry such as triangles, rectangles, hexagons, circles and so on. At a higher-level, an operator such as *polygon* may be available to construct assemblies of lines to represent a generic polygon. In the high-level notation, a polygon is defined to depend on a parameter representing its number of sides. Changing the value of this parameter directly influences the number of component lines in the assembly of the geometry at the lower-level. If definitions in the low-level script depend on the lines that represent a high-level polygon, what happens if the value defining the number of sides is redefined? This issue often creates a complex definition management problem, especially if the number of sides is reduced. The references that low-level definitions depended on for their evaluation may no longer exist.

The *polygon* operator is an example of *higher-order dependency*. Higher-order dependency and the different kinds of copy are compared in Section 3.2.3.

3.2.1 Definitions for Copying Assemblies

The process of making a copy of an object in a definitive script is very different from the style of *cut and paste* copying provided by tools such as word processors and drawing packages. The reason for this is that there is the need to consider the effect of dependency maintenance between the state of original assembly and its copied instance. An example of this in an application that contains dependency maintenance is the use of the *copy and paste* system for a block of cells in a spreadsheet. The default method for copying cells is one of many possible paths by which a copy could be carried out. Subsequent alteration to the values in these cells by a user may cause the update of other cells in a way that the user is not expecting.

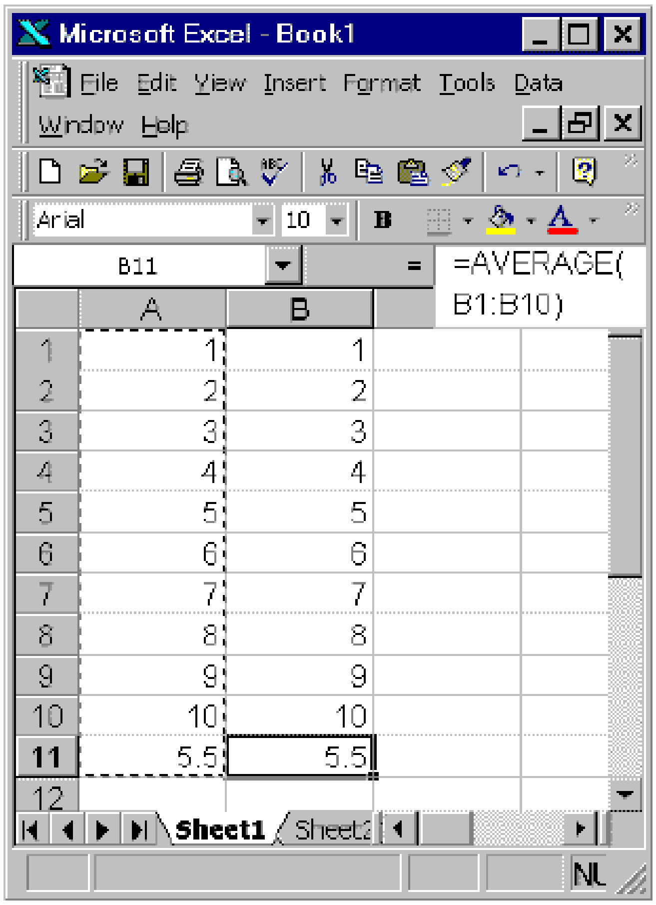


Figure 3.1: Copying a region of cells in an *Excel* spreadsheet.

Close examination of the traditional copying operation in a spreadsheet application illustrates some of the issues. Consider the spreadsheet shown in Figure 3.1. The first step in its creation was to enter the values in cells **A1** up to **A10**. Cell **A11** is then defined by a formula to be the average of these cells. The next step was to highlight cells **A1** through to **A11**, copy them and then paste them into column **B**. Cell **A1** contains only a number and not a formula, so only it's explicit value is copied. Any subsequent change to **A1** will not then update **B1**.

Cell **B11**, a copy of the formula in cell **A11**, is copied in a particular way. The formula from **A11** is not copied into **B11** verbatim, character by character. Instead, the range for the evaluation of the average formula is substituted with cells **B1** through to **B10**. Cell **B11** remains independent of cells **A1** to **A11**, even if the users intention for the copy is that cell **B11** is equal to the average of **A1** through **A10**. This would be the effect of an exact character by character copy of the formula. Many spreadsheet applications provide some form of a *special paste* that allow a user to set the future behaviour of a paste operation to suit subsequent interaction with a spreadsheet model[4].

A definition for an observable is either explicit of implicit. An *explicit definition* is of the form "*identifier* = *value*", where the identifier on the left-hand side is associated with the explicit value on the right-hand side. An *implicit definition* is of the form "*identifier* = *function*(*arguments*)", where the identifier on the left-hand side is associated with a value that is evaluated by applying the function to the sequence of arguments.

The process of copying a single definition that is for an explicit value, has only two possible outcomes in term of definitions to represent that copy in future states of the same script. Consider the example definitions "$a = 3$" and "$b = copy(a)$".

[4]The spreadsheet used for this example is Microsoft's *Excel 97* [KDS96, Jac97], which includes a "Paste special" operation to give a user additional control over copying of cells.

The definition of b is a new definition that a user passes to a tool for dependency maintenance that can implement the *copy* process for a in one of two ways, as shown in the two cases below. To name the different kinds of copy, analogies are drawn to mechanisms used to copy images in the real world.

Photographic copy The value associated with identifier b is set to be equal to the value of a at the moment of the copy. In this case, the definition "$b = copy(a)$" cannot remain in the script and should be replaced by "$b = 3$". This is similar to a photograph where the image in the picture remains exactly the same as the image viewed by a camera lens the instant that the shutter was open.

Mirror copy The value associated with identifier b depends on the value of a and subsequent change of the value of a changes the value of b. The definition of b is unchanged. This is similar to the way in which the image in a mirror continuously reflects the image of objects placed in front of it, including the motion of these objects.

When copying a single definition that is implicitly defined, there are at least three possible outcomes in terms of definitions to represent that copy. Consider the example definitions "$a = c + d$" and "$b = copy(a)$". Photographic and mirror copies for the implicit definition are similar in the list below to the types of copy for explicit definitions in the list above. Reconstruction copy is a more general case than the first two and can take more than one form. Reconstruction of an image involves the use of the same or apparently identical components of an original to make a copied instance.

Photographic copy The value associated with identifier b is set to be equal to the value of a in the state of the script at the point of the definition of b. For a value of $c + d$ of 4, the definition "$b = copy(a)$" should not remain in the script and is replaced with "$b = 4$". Subsequent change to the values of a, c or d does not affect the value of b.

Mirror copy The value associated with identifier b depends exactly on the value of a and any subsequent change to the value of a changes the value of b. The definition of b should remain unaltered.

Reconstruction copy The value associated with identifier b is defined by the same implicit formula as the value of a. The definition of b is replaced by the definition "$b = c+d$". Subsequent redefinition of a will not effect the value of b. Further interpretations of the copy of a are possible in this case, where implicit and explicit definitions are mixed in the right-hand side of the formula. For example, if the current value of d is 2 then the definition of b can be replaced by "$b = c + 2$". In this mixed argument example, the value of b is effected by subsequent change to c but not by subsequent change to d.

All the cases above are representations of copying processes that define the value of the copy to be identical to the value of the original definition at the moment that the *copy* definition for b is introduced into the script. The process by which the copy is represented in the creation of new definitions in a script determines the effect that subsequent redefinitions have in the propagation update of the current state of the values in the script.

With an assembly of geometry, there are several more cases possible than those listed for one single definition. If T is an assembly of some geometry, the definition "$S = copy(T)$" can have many interpretations for the definition of the subcomponents of the assembly. For every component definition of an assembly that is implicit, there can be any of the three types listed for single implicit definitions above. The effect of subsequent redefinition of the initial subcomponents on the copy is determined by the procedure for the creation of the copied instance by a tool for dependency maintenance. In general, it is better if every subcomponent definition is handled in exactly the same way to avoid ambiguity. It may, however, be necessary to consider mixtures of the procedures presented for copying subcomponent definitions in certain applications.

Figure 3.2 is split into six parts. Each part contains an identical piece of geometry defined by an assembly of definitions and the script to represent that geometry. A brief synopsis of the script notation used in the figure is presented in Table 3.1. The diagram of the geometry is labelled with parametrisations that correspond with the associated script of definitions and highlight the differences between different scripts the represent the same piece of geometry. The differences relate to the dependencies between the subcomponents of the assembly and other parameters in the script. In Figure 3.2a, the basic shape "`section`" represents the assembled geometry that is considered as the original geometry. This is copied in the other parts of the figure. This original geometric assembly consists of two lines ("`l1`" and "`l2`") and an arc ("`a1`"). The defining parameters of the geometry are a centre "`c`" and a radius "`r`". The subcomponent geometry of the *arcline* assembly depends on these parameters.

Each part of Figure 3.2 other than the top left hand corner represents a copy of "`section`" that is called "`section2`". These parts correspond to possible interpretations of the definition "`section2 = copy(section)`" and the script shown in each of these parts details the resulting definitions that persist in a script to represent the copy. The following list describes each part of the figure in more detail.

b - Every single subcomponent definition in the *mirror copy* defines the value of the associated identifier in the copy to be equal and dependent on the value of the identifier with the same name in the original assembly. Any subsequent

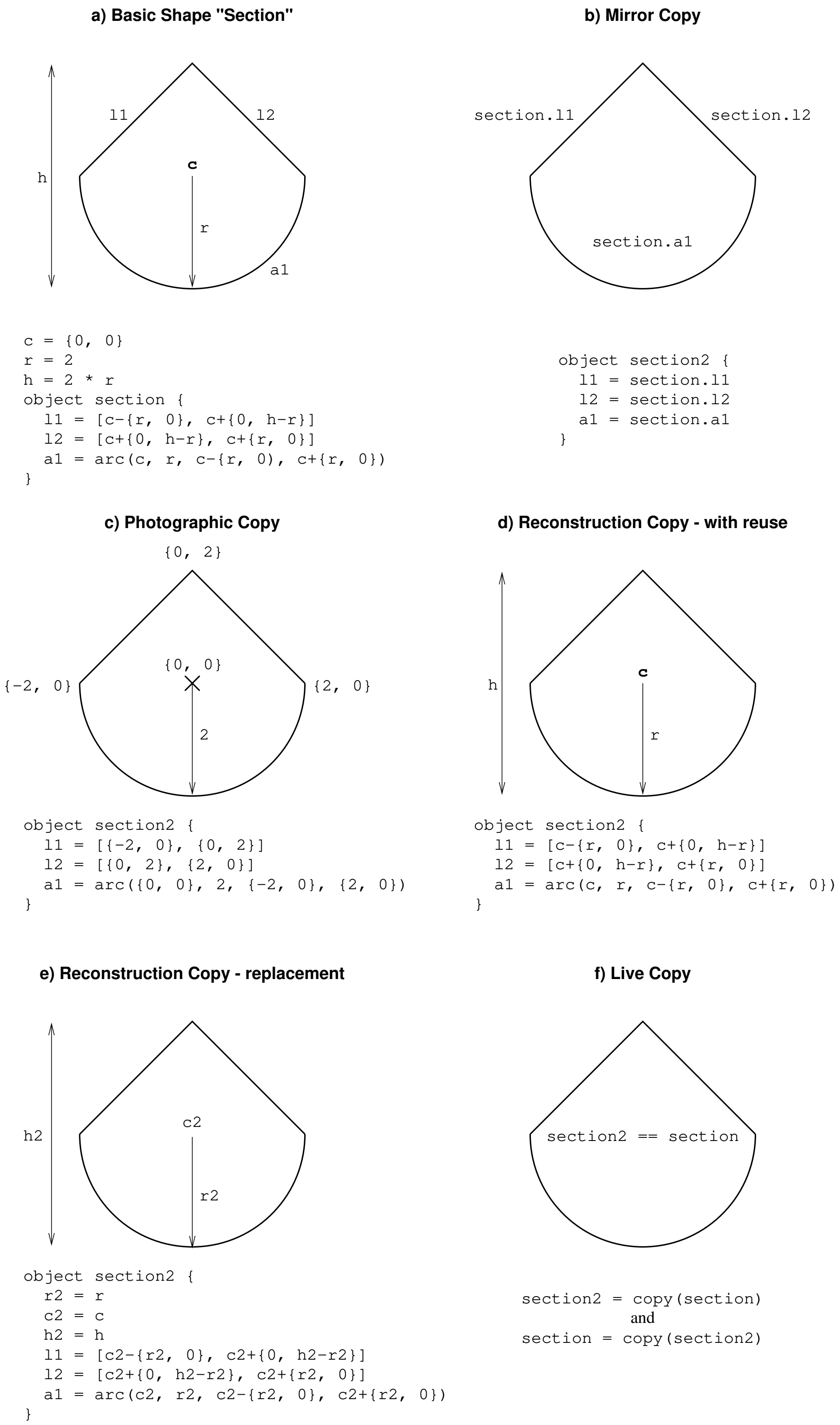


Figure 3.2: Different procedures for the copy of an assembly of geometric definitions.

Expression	Description
id = *expr*	Definition of identifier *id* to be equal to the right-hand side expression *expr*.
{p, q}	A point on the drawing plane defined by scalars `p` and `q`.
[m, n]	A line on the drawing plane with end points `m` and `n`.
arc(c, t, p1, p2)	An arc on the drawing place with centre `c`, radius `r`, start point `p1` and end point `p2`.
object o { ... }	An assembly of definitions called `o` constructed from a group of subcomponent definitions.
x.y	Reference to subcomponent `y` of assembly `x`.

Table 3.1: Synopsis of the script notation used in Figure 3.2.

redefinition of the original `l1`, `l2` and `a1` will propagate to effect the values associated with the copy.

c - The *photographic copy* defines the assembly `section2` to be equal to the values associated with the definitions in the original `section` in the state as they are in at the instance of the copy. Subsequent changes to the original shape or its defining parameters will not propagate through to update the copy instance.

d - For the *reconstruction with reuse copy*, subsequent redefinition of the original `l1`, `l2` and `a1` parameters does not propagate to the copy. However, the change of the value through redefinition of the defining parameters `c`, `r` and `h` will.

Other possible interpretations of a copy are illustrated in Figure 3.2e and Figure 3.2f. The *reconstruction with replacement copy* of Figure 3.2e is a variant of the *reconstruction with reuse copy*. The right-hand side of the definitions for `l1`, `l2` and `a1` of `section2` are similar to the right-hand side definitions for `section` with the parameters `c`, `r` and `h` replaced by `c2`, `r2` and `h2` respectively. At the exact moment of the copy, `c2` is defined to be equal to `c`, `r2` equal to `r` and `h2` equal to `h`. Subsequent changes to the parameters `c` and `r` will effect both the original (`section`) and the copy (`section2`). The definition of the defining parameters can be redefined to break their link with the original defining parameters. The copy can become independent of the original in this way. This reconstruction copy can be viewed as a way of creating templates of dependency that can be instantiated and customised by their parameters.

Figure 3.2f illustrates a script in which the high-level definition "`section2 = copy(section)`" is the persistent definition of the copy and there is no copy of the assembly of definitions. This copy is like the image on a television set of a live broadcast, where whatever happens in front of the camera lens in the television studio is shown on the television screen. In this case, there is no redefinable subcomponent geometry for `section2` as definitions in their own right. If there were, redefinition of a subcomponent would cause `section2` to be no longer consistent with its definition as a copy. This kind of copy does not fit well in current definitive programming environments. There are several possible ways in which subsequent interaction with the subcomponents of `section` and `section2` can be handled by a dependency maintaining tool. For instance:

1. Attempts to redefine the subcomponents of `section2` by the user cause a runtime error to be reported.

2. A redefinition of the subcomponents of `section2` is considered as a redefinition of the same subcomponent of `section`. Using this procedure, the definitions "`section2 = copy(section)`" and "`section = copy(section2)`" can coexist in the same script.

In this section, several different procedures for copying assemblies of geometry with definitions have been discussed. Note that if there were no assemblies

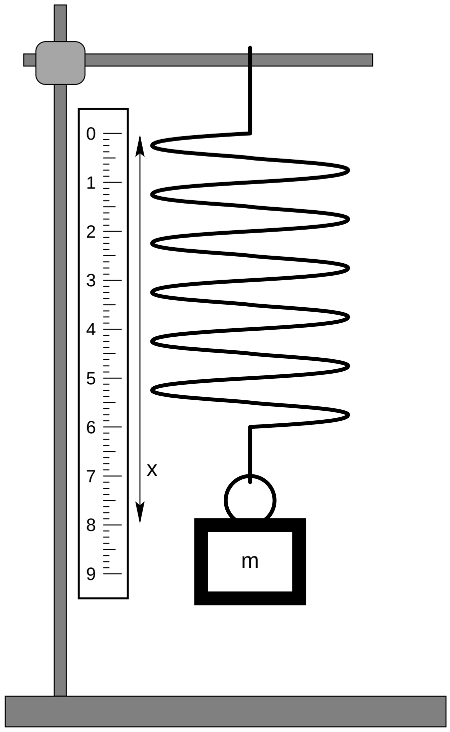


Figure 3.3: Mass on a spring experiment.

of definitions then there would be far fewer possible procedures for copying single definitions.

3.2.2 Higher-Order Dependency

The process of model making requires a period of observation of a real world phenomena and the recording of quantifiable data during a period of experimentation. This data can then be analysed, possibly through the preparation of a graph or other graphical representation, in such a way that patterns of behaviour that exist between certain observables can be determined. In the empirical modelling paradigm, this dependency between observables can be represented in a definitive script. Once the patterns between primitive observables have been represented in definitions, it may be that it is possible to observe patterns in the definitions that could be expressed in a higher-order definition.

For example, take the experiment that demonstrates Hooke's Law as shown in Figure 3.3. Increasing the mass increases the length of the spring and reducing the mass shortens the spring length. Experiments with several different springs can be carried out to measure the relationship between mass and extension. From the data set of each experiment there can be observed, within a certain margin for error, a proportional relationship between the mass and the extension. For each different spring tested, it is possible to express as a definition the observed relationship between mass and extension by finding an explicit constant for the spring concerned.

At some level of abstraction above the single spring on experiment, it can be observed that there is a common dependency to many spring experiments. Each has the same template definition describing the relationship between the length of the spring and the mass, the only difference is some constant of multiplication. This pattern between definitions can then be represented as a template definition describing all experiments involving masses suspended on springs.

This is similar to the *reconstruction with replacement* process shown in Figure 3.2 and described in Section 3.2.1 of the chapter. In this copying process, the defining parameters for a geometric assembly are identified independently of the dependency between the subcomponents and a local version of these parameters is created in the copied assembly. The equivalent in higher-order dependency is that these parameters become arguments to a high-level implicit definition, where the value associated with the left-hand side identifier of the definition is an assembly. With no original to copy from, the assembly is created by the high-level definition. The higher-order definition is in a separate definitive script notation at a level of abstraction above the representation of the low-level dependency for the geometric assemblies.

The table below shows high-level definition and the associated low-level ge-

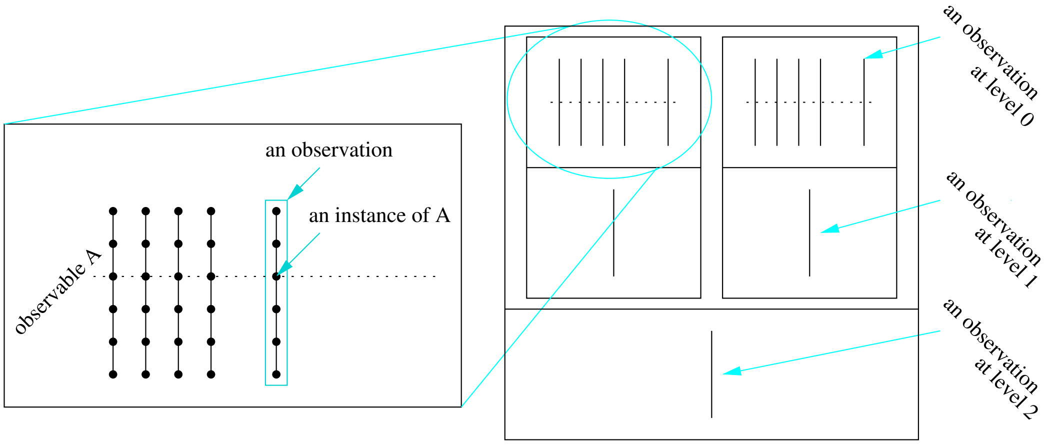


Figure 3.4: Levels of abstraction in observations.

ometric assembly generated by the high-level definition. In the high-level notation, the definition below of the arc and two line shapes called "*arcline*" will generate and maintain dependency for the assembly in the low-level script. The geometry of the *arcline* is the similar to the examples in figure 3.2. Redefinition of the value of r or the point c at the high-level will redefine the explicit values of `section3.r` and `section3.c` at the lower-level.

High-Level $\quad section3 = arcline(r, c)$

Low-Level
```
object section3 {
   c = Current value of c at high-level.
   r = Current value of r at high-level.
   h = 2 * r
   l1 = [c-{r, 0}, c+{0, h-r}]
   l2 = [c+{0, h-r}, c+{r, 0}]
   a1 = arc(c, r, c-{r, 0}, c+{r, 0}]
}
```

This process of observing phenomena from patterns in experimental data to form higher-level definitions at many increasing levels of abstraction is generalised in Figure 3.4. The advantage of characterising common patterns in observed real world data is that it is often more appropriate to represent these patterns rather than to model the original data sets in their entirety. The diagram shows three separate levels in the abstraction of higher-order dependency that are also listed below.

Level 0 Observation of data that may be related in some way, over the variation of one or more parameters. In the figure, a single straight line represents a snapshot of a system's state and the dots represent the value of the observables during the snapshot. In the Hooke's Law example the parameter being varied is the mass and the effect on the extension is observed to see if it is possible to establish a dependency between these parameters at level 1.

Level 1 Observation of data from level 0 shows that there is a dependency between certain observed values. This dependency is represented as one level 1 definition. In the Hooke's Law example, the mass applied and resulting extension for one spring experiment is represented as a definition.

Level 2 Observation of a number of dependencies shows that there are similarities between the definitions that represent common level 0 phenomena at level 1. These patterns of dependency also be observed and can be represented in one level 2 higher-order definition. In the Hooke's Law example, the level 2 definition represents the fact that there is a linearly proportional relationship between mass and spring extension in all such experiments.

Level n It is possible to extend the process of observing higher and higher levels of dependency between sets of definitions up to some finite level **n**. For the Hooke's Law example, the top level possible is level 2 as there is only one definition at this level.

This process, when used to construct definitions in a script, is an observation-oriented method for identifying structure in observed dependency. It is motivated by observation of phenomena that already exist in the world and can be experienced in some way, even if that experience is one of some new conceptual model that has never been realised as a real world referent[5]. In a geometric context, the process can be used as a means to identify and represent common parameters and varying parameters between geometry that has a common basis. For example, a rectangular box has a height and a width at *level 1* but consists of four lines in a geometric assembly of definitions at *level 0*.

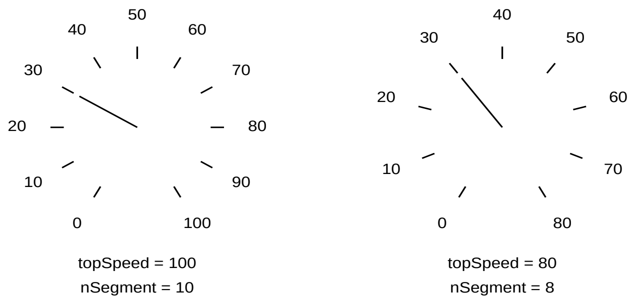
3.2.3 Comparison of Complex Dependencies

Many similarities exist between the many procedures for a copy in a tool that supports definitive scripts and the management of higher-order dependency. The two methods of structuring dependency for similar entities using assemblies and level of scripts are compared in this section. Higher-order dependency is a useful process for identifying patterns in real world observation for the construction of computer-based artefacts. Copying is a useful tool in the incremental construction of computer-based artefacts.

When a user interacts with a script, there are many ways in which a created object can be instantiated and linked to its original object component. The problems with many methods for copy, as detailed in Section 3.2.1, are associated with assemblies constructed from subcomponent definitions. Higher-order definitions allow a user to go from observation to reusable templates for making several lower-level definitions, whereas making a copied instance of an object is used as a tool in creation of objects in scripts. Tools that support empirical modelling with geometry should incorporate both support for higher-order template definitions and the ability to make copies of newly created definitions with a high degree of flexibility.

With higher-order dependencies, it is possible to create a template for geometry. This is similar to the way that a programmer creates a primitive object

[5]For further examples, see [GYC+96].




```
graph speedo
within speedo {
   real needleLength = 100.0
   real minA = 4 * pi div 3
   real maxA = - pi div 3
   real A = minA + (maxA - minA) * ~/curSpeed div ~/topSpeed
   line needle = [{0,0}, {needleLength @ A}]
   real gap1, gap2, LSpc
   gap1, gap2, LSpc = 10.0, 30.0, 50.0

   x<i> = ~/topSpeed * <i> div nSegment
   f<i> = minA + (maxA - minA) * <i> div nSegment
   nSegment = 8
   node = [
      label: label(itos(trunc(x<i>)), {(needleLength + gap2 + LSpc) @ f<i>});
      line: [{(needleLength + gap2) @ f<i>}, {(needleLength + gap1) @ f<i>}]
   ]
   segment = []
}
```

Figure 3.5: Two speedometer models and one template definition in DoNaLD that represents both models.

constructor that can be used in a drawing application, such as *xfig*. The generic template definition has a right-hand side that consists of parameters that will make the left-hand side of the definition reconfigure, where the left-hand object entity consists of assembled lower-level definitions. The example in the previous section for an "*arcline*" demonstrates how the defining parameters for the low-level dependency can depend on high-level values. The number of subcomponents of an assembly may also change depending on high-level defining parameters. Changing parameters on the right-hand side of the high-level generic definitions can cause the assembly of lower-level definitions to reconfigure.

One form of higher-order dependency is already implemented for the DoNaLD notation [ABH86], which is part of the *Tkeden* tool. It is known as the *graph abstraction*. Figure 3.5 shows generic template definitions in use in DoNaLD to describe the layout of a speedometer that has a top speed and number of division segments on the right-hand side. At the lower-level, there is a block of DoNaLD definitions representing the picture of a speedometer. Changing the number-of-segments parameter, considered as on the right-hand side of the high-level definition for the speedometer, leads to the reconfiguration of the low-level subcomponent definitions of the assembly of geometry.

The number of subcomponent definitions generated on the low-level left-hand side will change depending on the value of the special parameter "nSegments". The abstraction is considered as based on a two-dimensional plotted line with x values along the x-axis and $f(x)$ values along the y-axis. These are specified by definitions in the DoNaLD notation by descriptions of template definitions "x<i>" and "f<i>". Where "<i>" appears in these definitions, the elements of the sequence "0, …, *nSegments*" are used to create new low-level definitions for "x_0, …, x_*nSegments*" and "f_1, …, f_*nSegments*" respectively. The right-hand side of these definitions have the token <i> replaced by the index through the sequence for the definition. The "`node`" and "`segment`" definitions can be used to create a *mini script* of DoNaLD automatically for each segment of the graph, depending on the values of `x<i>` and `f<i>`. In the speedometer example, these mini scripts correspond to the speedometer graduation lines and labels.

The problem with the DoNaLD implementation is that if a user redefines a subcomponent definition then the graph is no longer consistent with its high-level definition. Maybe a designer wishes to change the length of the 70 miles-per-hour line segment to highlight the top speed limit on British roads. If the number of segments for the speedometer is increased, then this change will be lost and, in the worst case, there may no longer be a line segment corresponding to 70 miles-per-hour on the face of the speedometer. The order in which definitions are introduced into the script becomes important and the behaviour resulting from redefinition of parameters and the associated propagation of change introduces conflict. For one script of definitions there can be at least two possible states. The integrity of the state between the higher-level and lower-level of abstraction in scripts should be protected in some way.

The use of higher-order dependency in geometric modelling has a place once commitment can be made to the description of generic element templates through real world observations. In the creation of new geometry, these templates restrict the open-ended geometric design process. Higher-order abstraction in definitions is a tool that can be used for the constructing primitive geometric elements. Copying entities is a process that allows a user to take existing structure and reuse it without having to reconstruct of reason about dependencies between internal subcomponents at different levels of abstraction. The challenge of managing dependency between assemblies of definitions is to find a consistent and unambiguous way to support many kinds of copy and higher-order dependency in unified definitive notations.

3.3 Data Structure and Dependency

This section examines the issues of integrating arbitrary data structure and dependency into the same definitive notations. High-level programming languages provide data types for the representation of atomic data such as scalar and Boolean values. These atomic data types can be combined into arbitrary new data structures with *abstract data types* (ADTs) [AHU82] that represent data that is more complex than a variable of one atomic type can represent alone. The structure can be given an identity in its own right and mechanisms exist for accessing components of atomic or other constructed data types. Data structure is an association of component data types that can be regarded as a form of dependency. Definitions in definitive scripts represent indivisible relationships between variables, considered as observables, of any data type. In existing notations, definitions can record structural dependencies (as in the point constructor "`p = {x, y}`") but not reflect their special characteristics.

For example, consider the case-study of assemblies of geometry in Section 3.2. Data types exist in the example notation in Table 3.1 to represent scalar values (integer or floating point), two-dimensional points (two scalar values) and two-dimensional lines (start and end points). These are the data types, and every variable of the non-atomic data types has an internal structure: points are constructed from the atomic data type for scalars, lines are constructed from component points. These types are provided as an integral part of the notation and in order to represent more complex data a user must create assemblies that group definitions for values of these types. Dependency can be established between single variables of the built-in data types at different levels in their internal data structures, where the value of each component of data is either explicitly given or implicitly given by the evaluation of the right-hand side of the definition.

The level at which the definition of a variable of a non-atomic data type is constructed, rather than given by definitions, is known as its *mode*. The ARCA notation [Bey83, Bey86a] for scripts of definitions that describe Cayley Diagrams includes an implementation of *modes* for its variables. The process of assigning a mode to ARCA variables is known as *moding*. The relationship between levels in data structure and dependency is considered through the examination of moding and ARCA in Section 3.3.1.

Data structure describes the association of component values within a compound data type. Dependency in definitive scripts represents indivisible relationship between variables. In Section 3.3.2, the argument that data structure and dependency can be regarded as orthogonal concerns is presented. The possibility of representing one dependency between variables of compound types as several dependencies between their components at an atomic data type level is demonstrated. The ambiguities introduced by dependencies that are expressed by definition between variables at different component levels in data structures are discussed.

3.3.1 Moding and ARCA

ARCA was the first example of a definitive notation to be developed at Warwick [Bey83, Bey86a][6]. The data types and operators of ARCA are illustrated in Figure 3.6. This shows an ARCA representation of the symmetric group S_3 in three sections. From top to bottom in the figure, these are: a Cayley diagram for S_3, a graphical representation of its ARCA data structure and two segments of its script of definitions in the ARCA notation. The explicit definitions for the diagram can be subsequently redefined to experiment with the state of the diagram. In the tool developed by Bird that implements ARCA [Bir91], the screen image of the group reconfigures to reflect the current state of the definitions and their associated values

[6]Earlier work on definitive notations can be traced back to Brian Wyvill [Wyv75].

(see example ARCA output in Figure 2.1).

The main compound data structure in ARCA is the *diagram* that represents the coordinate location of the N component *vertices* in a diagram and the partial injection mapping from $\{1, \ldots, N\}$ to itself associated with edges of each *colour* in the diagram.

A variable (`x`) or a component of its structure (`x[1]`, `x[2]`, ..., `x[n]`) can be declared to be in either *concrete* or *abstract* mode. The significance of these modes is as follows:

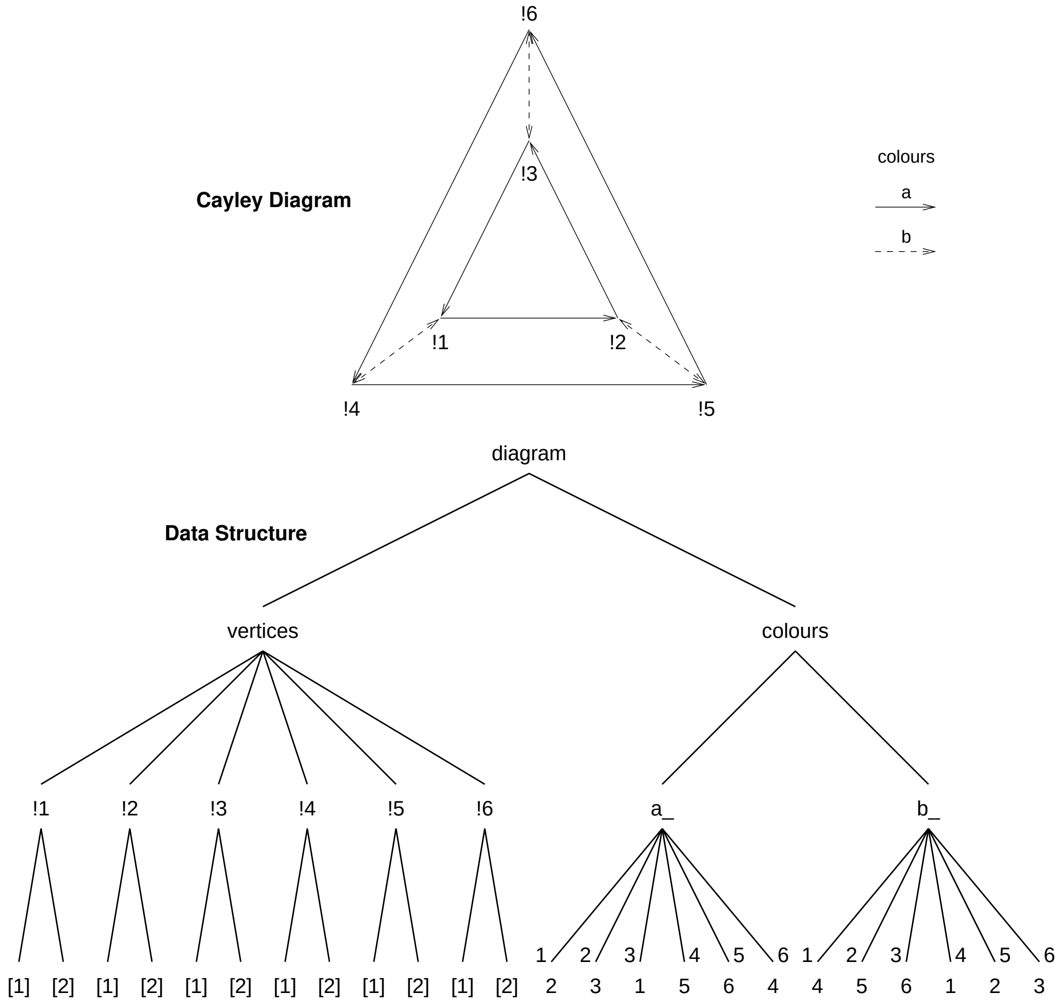
abstract mode The value of the variable or component is given by a constructor at the same level or defined by an implicit definition. It is not defined component-wise.

concrete mode The value of the variable or component is constructed from the definitions of its components.

In ARCA, if a variable is declared to be in abstract mode, then no declaration is required for the mode of its components. If a variable is declared to be in concrete mode then the mode of its components must also be declared.

The data structure for the diagram in Figure 3.6 is in concrete mode and all component values are in concrete mode down to its leaves. The start of the example script shows combined mode and type declarations for the identifiers to be defined by explicit values further through the script. A mode declaration of the form "`mode x = 'ab'-diag 6`" describes `x` as a diagram with $N = 6$ vertices and two colours `a` and `b`. Declaration of the mode of the component vertices and colours is then required.

Declaration of the dimension of each concrete ARCA vertex is of the form "`mode y = vert 3`", describing that the value of `y` is constructed from the values of its components `y[1]`, `y[2]` and `y[3]`. These components can be implicitly or




```
mode dia = 'ab'-diag 6
mode dia!1 = vert 2
mode dia!2 = vert 2
mode dia!3 = vert 2
mode dia!4 = vert 2
mode dia!5 = vert 2
mode dia!6 = vert 2
mode a_dia = col 6
mode b_dia = col 6

⋮
```

```
a_dia = {1,2,3}${4,5,6}
b_dia = {1,4}${2,5}${3,6}
dia!1 = [-200, -150]
dia!2 = [200, -150]
dia!3 = [0, 250]
dia!4 = [-400, -300]
dia!4 = [400, -300]
dia!6 = [0, 500]

⋮
```

Figure 3.6: An explicit ARCA diagram for the symmetric group S_3.

explicitly defined. The declaration of an abstract ARCA vertex takes the form "`mode z = abst vert`". This signifies that `z` is defined explicitly by a constructor that creates a vertex, or defined implicitly to be dependent on other vertices or scalar values. With this declaration for variable `z`, its value cannot be established by the definition of its components.

For every variable in ARCA, there are levels ar which its components are in abstract or concrete mode. These levels can be considered as establishing a moding template for definitions. These templates allow a user to establish dependencies between components that would otherwise be considered as cyclic dependencies (see Section 4.2.3). For example "`x[1] = x[3]`" is not a cylic dependency with concrete mode variable `x` constructed from components `x[1]` and `x[3]`. In contrast, an abstract mode variable `x` with definition "`x = [3, 2, x[1]]`" does lead to cylic dependency as the evaluation of the right-hand side depends on itself.

The ARCA notation treats the mode and type declarations as if they were definitions. In effect, the moding of variables is handled by an auxilary definitive notation. In theory, it is possible to redefine them and hence affect the mode and data structure associated with identifiers in a script through indivisible propagation of change. In practice, it is technically difficult to implement a system that can handle radical redefinition of the type associated with an identifier. For example, an implementor of the ARCA notation has to consider mechanisms for handling the effect of the interactive redeclaration of a variable of diagram type to be of vertex type, or redeclaring an existing abstract mode diagram to be concrete (so as to allow dependencies between its components to be specified).

In a high-level language, the underlying algebras over the atomic types include a look-up table for each operator. This table determines the type of the value returned by the operator, which depends on the types of the arguments. Type checking is performed during compilation of code to check that the data types declared to be associated with variable identifiers are consistent with those returned by right-hand side expressions. In ARCA, built-in operators in the notation exist that can be used in definitions to define indivisible relationships between values of compound data types. This requires that every operator has a look-up table for return type and its return mode.

Every variable in a definitive script such as ARCA has an associated mode and every expression in the script also has a mode. The reasons for moding expressions are:

- to attach a structure to values returned;
- to ensure this structure is consistent with the left-hand side variable in a definition.

For example, consider a concrete mode list `l` with three elements, declared as follows:

```
mode l = list 3
```

Consider also a general list operation `reverse` for list reversal. The mode of the expression `reverse(k)` is abstract, as the operator returns a list with a different number of component elements depending on its argument `k`. Due to the varying length of the returned list, it is not appropriate to define the three element list `l` by the definition

```
l = reverse(k)
```

or by the group of definitions

```
l[1] = (reverse(k))[1]
l[2] = (reverse(k))[2]
l[3] = (reverse(k))[3]
```

The concrete mode of the operator `reverse3` for reversing the order of three element lists is `list 3` $\rightarrow$ `list 3`. In this case, the mode definition "`l = reverse3(k)`" is appropriate as the returned value always has three components. Notice how the mode of a variable also affects the nature of a reference its value on the right-hand side of another definition. If a variable `v` is in abstract mode then it is possible to refer to the third component of the value of the variable `v` (through a projection operator `p3(v)`) but not to the value of the third component of the variable `v` (by `v[3]`).

Moding is required in a definitive notation with compound data types. The ARCA interface to moding can be improved to better support moding outside of the script notation. This improvement is hard to implement, especially to support the the degree of flexibility envisaged. The machine models developed later in this thesis handle this well and do more towards supporting user-defined data types and operators than EDEN. Existing tools are not good for implmentations that support moding, with particular problems posed by the representation of EDEN lists (discussed in Section 3.4.1). A solution is needed (such as will be described later) that not only allows user-definition of underlying algebra, but also deals with moding to accompany this.

3.3.2 Orthogonality between Data Structure and Dependency

In a definitive script, there can be dependency between components of the data structure for a variable and dependency between variables. In this section, the relationship between dependency and data structure is considered. It is appropriate to represent data structure and dependency as if they are orthogonal concerns. Every identifier and component reference in a script of definitions can have both a location in a data structure and be dependent on other data values through definition. A value is made up from component parts if it is of a compound data type and, as discussed in Section 3.3.1, can be defined in different modes.

In general, it is possible to find a way of representing all dependency given at non-atomic data type level as dependencies between component values of atomic types. If all dependency is represented at the atomic type level, then there is no ambiguity introduced as to which relationships are dependency between values and which relationships exist for data structure. All structure in dependency is between scalar values in the level of the atomic data types and all compound data types are constructed from these atomic values to form levels in data structures for high-level vatiables. In this way, data structure is considered in this thesis to be orthogonal to dependency by representing dependency at the scalar data structure level.

For example, Figure 3.7 shows a representation of the levels of data structure for a two-dimensional straight line data type. The line is represented by two component end points and each of these points is represented by a pair of scalar values. These scalar values are atomic types in the example. The figure is separated into three sections and the value of line variable `l` is the same in each. Figure 3.7a shows a line that is explicitly represented at all levels. Figure 3.7b and Figure 3.7c show the line with components defined implicitly.

Dependency between data of the same type at the same level is diagrammatically representable as embedded in that level. In Figure 3.7b and Figure 3.7c, it is possible to extract and describe a low-level of dependency between atomic scalar data types, another level of dependency for point compound data types and another for line compound data types. Dependency can also exist between data types at different levels. For example, an inner product operator maps two point arguments to a scalar value. The linking of different levels in data structure by dependency in this way leads to an ambiguity in the possible combined representation for the data values and the dependencies between them as variables in a script. The same dependency is shown represented in Figure 3.7c as in Figure 3.7b, except that it is

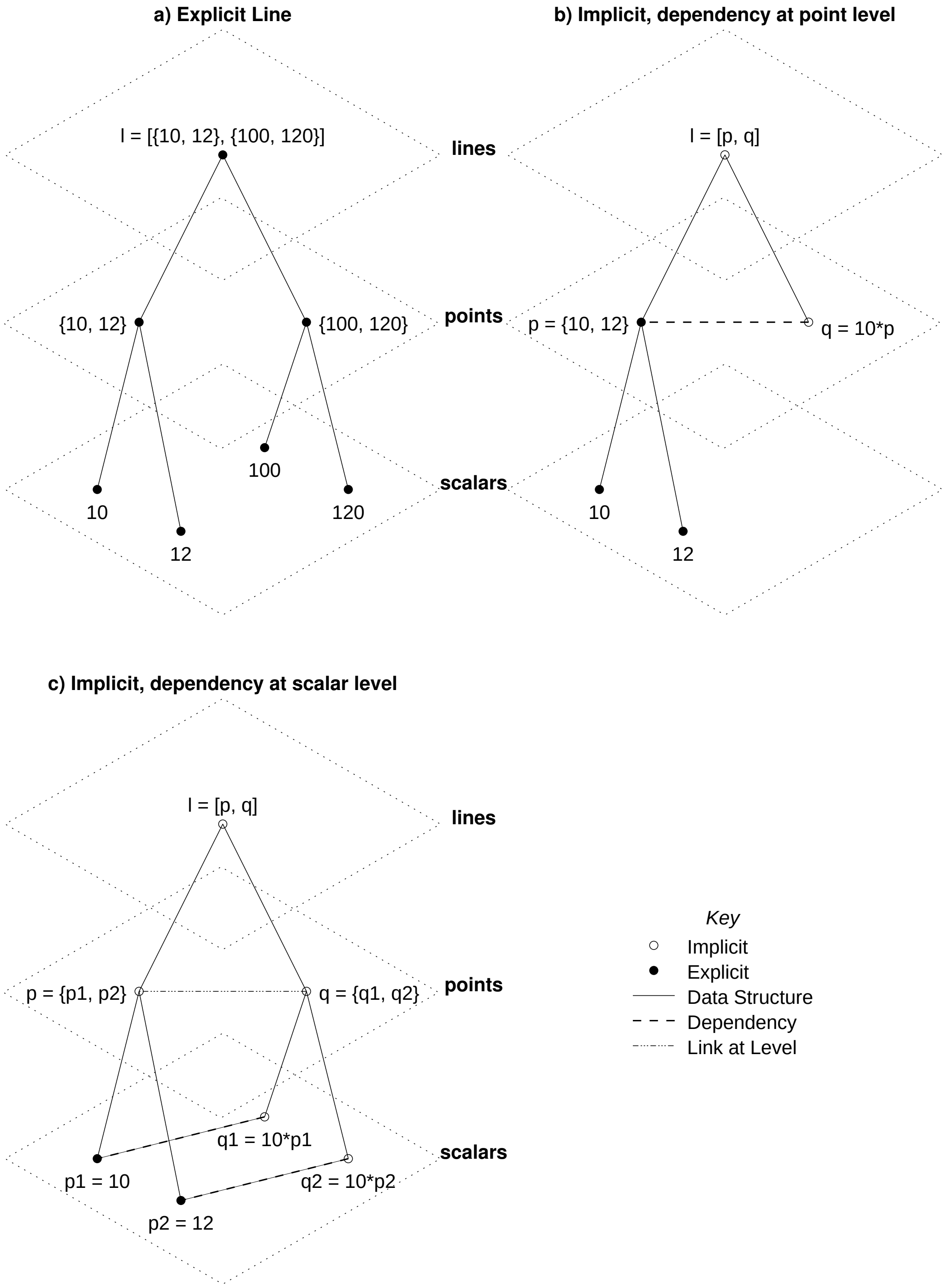


Figure 3.7: Levels in data structure and dependency for the definition of a straight line.

represented at the scalar level rather than the point level.

Data structure can be constructed by implicit definition. For example, a two-dimensional point value is represented by a data structure with component scalar values. A variable for a point can be constructed by implicit definition with a defining function `makePoint` that maps from two scalar values on the right-hand side to point variable on the left-hand side. In this case, it is not clear whether the point is a variable constructed in concrete mode or is implicitly defined. It is still possible to handle dependency between components in this case without introducing cyclic dependency. For example, consider the following definition for constructing point `p` from components `p1` and `p2`:

```
p = makePoint(p1, p2)
```

A dependency between `p1` and `p2` cab be expressed with a definition such as "`p1 = 2*p2`". A high-level definitive notation with support for moding can be represented at a lower-level by operators for constructors and dependencies between component definitions.

3.4 Data Representation and Dependency

In this section, methods for data representation for observed values that are appropriate to definitive scripts are considered, with a particular focus on geometry. To put this discussion in context, existing methods for the representation of geometric data in existing tools are presented. This is followed by a discussion of data where the parametrisation of entities is of a discrete nature but the actual representation of the value is apparently continuous and can be sampled at arbitrary values. Functions that are used on the right-hand side of definitions to create indivisible relationships between entities can operate over internal parametrisations of the entities and from a *continuous* representation of the entity (such as a function representation of its shape). Section 3.4.3 considers additional types of data that may be associated with geometric entities, such as graphical attributes, that ideally need to be represented in a definitive notation for geometry.

3.4.1 Existing Data Representations

The EDEN tool has built-in support for some atomic data types and one compound data type. The atomic data types represent integer values, floating point values, characters, strings of characters and a special type called *undefined* (represented by symbol "`@`"). The tool supports dynamic typing. The type associated with a variable is determined by interpreting the script of definitions and inferring types to be associated with identifiers on the left-hand side of explicit definitions from the string of characters on the right-hand side. For a implicit definition, there is a lookup table for each operator that determines the type associated with a variable from the arguments to the operator. The redefinition of a variable of integer type to be of string type may cause implicitly defined variables with numeric expression definitions to become undefined, the one and only value of the undefined type. Any other value that depends on an undefined value can itself become undefined.

This process is in one way very powerful as there is no need to declare variables before use to be of one particular type. The system can make sensible changes to types of variables and propagate type change through a script of definitions. A modeller using the EDEN notation benefits from this process, as they can concentrate on interactively constructing the model and identifying the best observables for a particular model without worrying about the types of the variables used to represent the observables. The redefinition of a variable that causes another implicitly defined value to become undefined is not reported to the modeller and the process of establishing where the relationships between variables cause a value to be undefined can be time consuming.

No facility exists for the creation of abstract data types in EDEN, DoNaLD or SCOUT. A *list* type exists that can be used to group integer, floating point, string, character and other list values. Elements of a list can be references and set by an index value inside of square brackets "`[]`" and the whole list can be referenced and defined by a comma separated list of implicit and explicit definitions. For example, consider the definition of the list `l1` shown below.

```
r = 10;
l1 is ["circle", 2*r, 10, 10.5];
```

The list `l1` has four elements and is used to represent the parameters of a circle shape in two-dimensional space, where the second argument is the implicitly defined radius that depends on the current value of `r` and the third and fourth elements represent the centre point. The centre point in this example is explicitly given by an integer value (`10`) and a floating point value (`10.5`). To declare that the list represents a *circle* shape, the first element of the list is set an explicit definition of the string of characters "`circle`". The EDEN parser establishes that the first element of the list is explicitly given and of string data type, that the second element is implicitly given and currently of integer data type and so on for all the elements.

What is the effect of the following redefinitions of the list `l1`? Redefinitions of lists and elements of lists may not result in the effect that a modeller expects.

1. `l1[2] is 3 * r;`
2. `l1[2] = 3 * r;`
3. `l1[3] is l1[2];`
4. `l1 is ["circle", 2*r, l1[2], 10.5];`

The first redefinition (1) fails with an error reported to the modeller because it is not possible to implicitly define only an element of a list is EDEN. The second redefinition (2) succeeds as it is an explicit definition that takes the current value of `r`, multiplies this value by three and assigns this as the explicit value of the second element of the list. This redefinition actually alters the definition of `l1` without the modeller explicitly requesting this redefinition. The third redefinition (3) fails with an error reported for the same reason as the first redefinition. It is not possible to introduce a dependency between elements of a list in this way, or in the declaration of an entire list as shown in the fourth redefinition (4). The EDEN interpreter regards the definition of a list with a dependency between its elements as a cyclic dependency.

The only way to identify the type of data in a list is by placing some marker such as the "`circle`" string in the list. Operators in EDEN are called *functions* and are defined on-the-fly by sections of interpreted procedural code. The operators to these functions have their types determined automatically. In an application that represents geometry, the use of operators over geometry represented in EDEN lists requires that the operator type checks its arguments to see if the arbitrary list passed as an argument contains the expected parameters.

It is difficult to represent the assemblies of definitions introduced in Figure 3.2 using lists in EDEN because of the problems of establishing dependency between elements of lists. One of the purposes of grouping definitions in assemblies is to represent dependencies between components of an entity, and there needs to be another way to represent these. The DoNaLD notation, implemented as part of the *Tkeden* interpreter, uses EDEN as its *back-end* dependency maintainer tool. All DoNaLD definitions are translated into EDEN and assemblies of definitions in DoNaLD known as *openshapes* are represented in EDEN by variable naming conventions.

Table 3.2 shows a DoNaLD script of definitions and some EDEN definitions that represent the DoNaLD script. The openshape construct is used to create an assembly of definitions. In the table the assembled definitions are `l` and `i2`. All DoNaLD definitions that are not part of an openshape are preceded by an underscore

DoNaLD Definitions	Translated into EDEN Definitions
`int i`	
`i = 10`	`_i is 10;`
`openshape test`	
`within test {`	
`  int i2`	
`  i2 = 20`	`_test_i2 is 20;`
`  line l`	
`  l = [{~/i, ~/i}, {i2, i2}]`	`_test_l is line(cart(_i, _i),`
`}`	`cart(_test_i2, _test_i2))`

Table 3.2: Translating DoNaLD openshapes into EDEN.

character "_" in EDEN and for an openshape called "`x`", all definitions are preceded by "`_x_`" in EDEN. In this way, dependency between assembled definitions can be represented in EDEN. However, because the current version of EDEN has no support for the automatic manipulation of this naming convention consistent with some higher-level script, the DoNaLD translator must retain a record of the EDEN names.[7].

3.4.2 Parametrised Data Sets

Conventional computer systems are finite machines that are good tools for the representation and manipulation of discrete data. When continuous parameters, often analogues of real world observables, are represented on a computer system, they need to be digitised from the measured analogue parameter and represented discretely within a margin for error. Idealised geometry in Euclidean space is of a continuous nature and can only be represented on a computer system within certain margins for error. This section examines data values represented by parametrised data sets where instead of storing discretely representable value, a membership condition for the set is used to determine the value by sampling. The modelling of geometry of a continuous nature in a definitive script relies on good data structures

[7]Pi-Hwa Sun is currently working on introducing control in EDEN for this style of grouping of agents. His method is known as *virtual agency*.

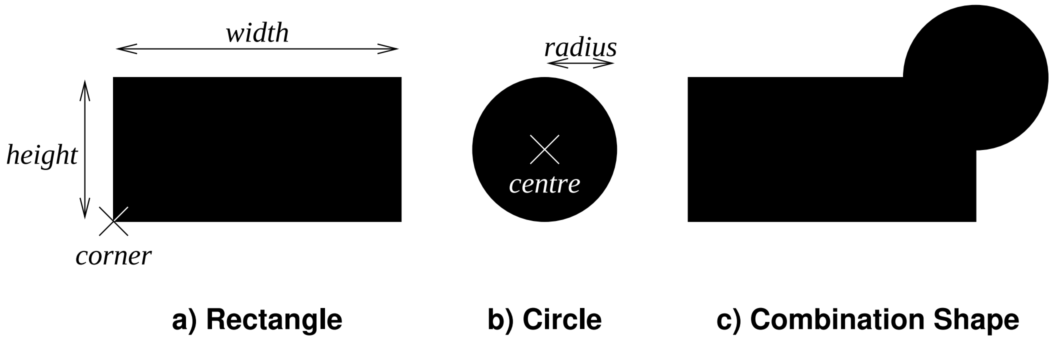


Figure 3.8: Parametrised shapes and their combination.

and mechanisms for the representation of parametrised data sets.

The solid two-dimensional geometric entities shown in Figure 3.8 are examples of sets of data of a continuous nature. Figure 3.8a is parametrised by a *height* value, *width* value and a corner point. The circle in Figure 3.8b is parametrised by a *centre* point and a *radius*. For both shapes, it is possible to implement algorithms that will draw graphical representations of the shapes on a computer screen and other procedural function code to establish dependencies between the parameters and the shape by definition.

The shape in Figure 3.8 is a combination of a solid rectangle shape and solid circle shape. One parameter-level method for representing the combination of shapes in definitive notations involves:

- describing parametrisations for combined shapes made from one rectangle and one circle shape and all combinations of shape primitives;
- implementing specific algorithms to render each of the combined shapes;
- establishing dependencies between the parameters for combined shapes and the parameters of their defining primitives.

This method allows for the construction of preconceived parametrised shape primitives but not for the arbitrary combination of any shape. The combined shape of Figure 3.8c can also be regarded as representing a point set union. A desirable feature in definitive script notations for geometric modelling is the inclusion of operators for the combination of arbitrary shapes, where the dependency is established between non discrete point sets rather than discrete defining parameters[8].

The CADNORT tool, developed as part of my third year undergraduate project [Car94b], translates scripts of definitions for two and three dimensional solid geometric shapes into EDEN definitions. For every shape description in the CADNORT notation, there is a block of EDEN definitions representing the parameters for that shape and an associated procedural function that tests for membership of the point set for the geometry. A similar naming convention to that used by DoNaLD and shown in Table 3.2 is used to represent the parameters of shapes in the EDEN model rather than a list representation.

In CADNORT, the function representation of shape [PASS95] is used as the mathematical basis for determining point set membership for the shapes. For any point in n-dimensional Euclidean space, point set membership is determined by a function f that maps from any point to a real value. To test point membership of a shape S at any point $\mathbf{p}$, the solid geometric shape is represented by function f, where

$$f(\mathbf{p}) \begin{cases} < 0 & \mathbf{p} \text{ is outside } S. \\ = 0 & \mathbf{p} \text{ is on the surface of } S. \\ > 0 & \mathbf{p} \text{ is inside } S. \end{cases} \tag{3.1}$$

Figure 3.9 shows the CADNORT graphical output for a table with a lamp placed on its top. The figure shows both front and side views of the sampled geometry at particular planar slices through the shape. The output shows only the outline of the solid shape. The generation of graphical output in CADNORT is a slow process due to the evaluation of interpreted code that represents the function

[8] DoNaLD is a notation for line drawing and does not attempt to represent filled shapes. Shapes can be displayed as filled by using attributes for the rendering of the geometry.

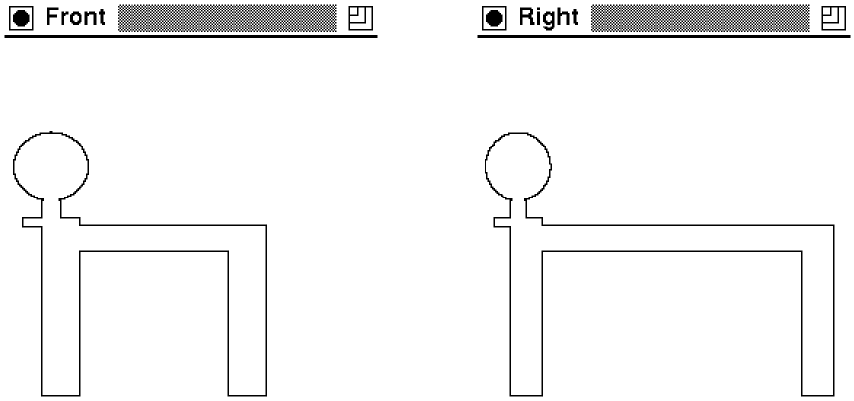


Figure 3.9: Graphical output from the CADNORT tool for a table and lamp script.

representations. The graphical output requires sampling of the function representation for the shape at every pixel. The dimensions of the graphics shown are 200 by 200 pixels and the generation of the figure was so slow in 1994 that plotted points could be observed to appear one by one.

CADNORT has some limitations. These include:

1. Managing the scripts of definitions in CADNORT and EDEN. The high-level definitive script for the table is approximately eight times shorter than the length of the EDEN script used to represent it.

2. Templates (called *generators*) representing families of geometry can be created on-the-fly. These templates cannot be redefined on-the-fly.

3. It is not possible to reference the components of shapes constructed using set-theoretic operations.

For all its limitations, CADNORT demonstrates the representation of point sets by definitions, with indivisible relationships established between point sets.

3.4.3 Geometric Data and its Associated Attributes

Large volumes of data are associated with representations of geometry on a computer system. Computer-based tools to assist a designer to model geometry typically represent geometric entities together with their defining parameters. There is a wide diversity of data that may also be associated with each model. This can be represented in additional data fields in the data representation of the geometry. The kinds of field that may need to be considered include:

Attributes Data that can describe non-geometric information associated with a geometric object. This can include the objects colour, texture, material, density, line width.

Location In addition to relative point locations within a geometric entity itself, the entity may be transformed into a different location before it is rendered. Geometric shapes are often placed in virtual computer-based worlds with other geometric shapes. The location of a shape in such as world may be different from the location where it was originally constructed.

Bounding Box Solid geometry, which has an associated point set that occupies a volume of space, may be boundable by a containing rectangle if defined in two dimensions, by a containing box in three dimensions, and so on. The bounding shape in any dimension is known as the *bounding box* and has sides that are parallel to the axis of the space. The bounding box is often required to assist algorithms for rendering geometry. Such a bounding box is particular useful for rendering shape represented by function representation.

Structure A geometric entity may itself be a child component of another entity or be the parent entity for other sub-entities. If this relationship is more than just a dependency given in a script, then it may be necessary to store information about the parent/child relationships with the data representation of a geometric entity. For example, there may be data concerning how the colours of entities appear to blend together when they are joined.

Reference A geometric entity contains information that may be useful for constructing the parameters of other entities. For example, a circle shape defined by centre and radius parameters may have an implicitly determined diameter parameter that can be used in the definition of other shapes that depend on the definition of the circle.

When creating a geometric model in a definitive script notation, it is desirable that a modeller should be able to modify not just data relating to defining shape, but also attributes, overall location, bounding box, structure and reference. The user should be able to make novel definitions between this data also, such as the colour of an object changing dependent on permissions to modify it. The challenge is to provide a data representation for geometry that not only handles defining parameters, point sets and the information in the above list, but is also able to handle the effect of the extended data representations when instantiating copies or forming high-order definitions.

Support for attribute data in existing definitive notations for geometry is poor. Attribute data in DoNaLD is not held as part of the geometric entity but instead relies on the underlying EDEN implementation of the DoNaLD interpreter to insert attribute information for use by the drawing actions of the interpreter. It is, for example, possible to link the colour of a line to its length but this has to be done at the EDEN interpreter level rather than within DoNaLD itself. Attributes cannot be applied to a group of definitions simultaneously and must be applied one by one. When making a copy of a group of definitions in an *openshape* in the form of a translation or a rotation, the attributes are not copied and it is not possible to assign different attributes to the components of the openshape.

3.5 Proposed Solution to the Technical Issues

The previous sections of this chapter have introduced some of the technical issues associated with representing data and manipulating data with scripts of definitions. Although these are general issues for empirical modelling, they pose particular problems when associated with geometry. The aim of the work presented in the chapters that follow is to develop a way of using definitive notations for geometry that overcomes these problems. An overview of the underlying approach adopted in this research follows.

The following list summarises some of the technical issues:

- The ambiguity in the status of copying definitions in a script occurs when copying an assembly of geometry rather than a single definition. Two copy procedures (photographic and mirror) exist for single explicit definitions and three classifications of copy procedures (photographic, mirror and reconstruction) exist when copying implicit definitions.

- Issues concerning the mode of definiton and the level at which dependency is defined in data structure arise when values are represented by data types other than atomic data types. Only one abstract mode exists for variables of atomic data types, which are either explicitly or implicitly defined at the same *scalar* level.

- Functions are required for definitions that represent indivisible relationships between:

  - defining parameters for geometric entities;
  - the point sets represented by geometric entities.

The above discussion motivates the introduction of a special "atomic" data type that can be used to represent all the explicit values associated with variables in a definitive script. The term *serialisation* will be used to refer to the process of transforming an explicit value of any data type into an encoded atomic value. If all values referenced in a definitive script notation (including all parameters, attributes and point inclusion algorithms and assemblies of definitions) can be represented by *serialised data types* in this way, many of the issues raised in this chapter are greatly simplified. Every value is either implicitly or explicitly defined and all explicit values are on the same data structure level. Higher-order dependency is a method for creating a lower-level script of assembly of definitions. If a value of a serialised data type can be used to represent all the information for an assembly, reference can be made to what previously would have been the subcomponent definitions of an assembly through the use of a special operator. These special operators map from a value of a serialised data type to a representation of a component its value. A low-level script is not required. For example, consider the *arcline* higher-order definition shown in Section 3.2.2. Table 3.3 shows atomic data types and operators that can be used both to create an arcline shape and make reference to its internal components. The script of definitions in Table 3.3 illustrates the construction of an arcline and reference to its component arc shape (*a1*).

Single values of serialised data types will need to be interpreted as representing a wide diversity of data, including sets. The operators that create indivisible relationships between these values will need to be able to extract and relate many different aspects of the data represented. As every value incorporates all information required for a particular type of data, interpretation of this data can include methods for determining set membership. These methods can be related to one another through special operators used in definitions in the same way as defining parameters.

Atomic Data Types	*scalar*	Floating point values.
	point	Points in two-dimensional space.
	arc	A curved section of a circle.
	arcline	A shape formed from an arc and two lines, as shown in Figure 3.2.
Operators	*makeArcline*	Define an *arcline* dependent on a scalar and a point.
	arcFromArcline	Define an *arc* to be equal to the arc of the *arcline* that is an argument to the operator.
Example Script		$$\begin{aligned} r &= 2 \\ c &= \{0,0\} \\ al &= makeArcline(r,c) \\ a1 &= arcFromArcline(al) \end{aligned}$$

Table 3.3: Replacing higher-order dependency with atomic data types.

In this way, all data structure, expression trees and set representation for a model can be represented by a script of definitions. The atomic types of the notations are similar to objects in object-oriented programming (OOP) [CAB+94]. Every object represents a *thing* through its data fields and its relationships to other *things* through its methods. The difference from OOP is that the values are defined in interactive, open-ended scripts and communication between values occurs through indivisible relationships established by definition. In OOP, the communication between objects is given by the procedural ordering of the methods and is prescribed at the time that the code is compiled.

In the next chapter, the potential for implementing definitive notations to handle serialised data types and operations is assessed, by examining dependency in scripts that have only one data type. This concentrates on issues that are related only to efficient dependency maintenance. This model is used as the basis of an implementation of dependency maintenance that only implements one atomic data type and requires a programmer to make decisions on how to organise the data of that type in a useful way (the DAM Machine in Chapter 5). The model is also the basis for an object-oriented implementation for dependency maintenance that supports the programmer in the implementation of serialised data types and special operators (the JaM Machine API in Chapter 6). For both implementation methodologies, the case-studies presented are based on geometric modelling examples.

In Chapter 9, the technical issues described in this chapter are reviewed with resepect to the special atomic data type approach to dependency maintenance.

Chapter 4

The Dependency Maintainer Model

4.1 Introduction

Dependency maintenance can be carried out at many levels of abstraction without compromising semantic subtlety of definitive state representation. Data structures can be refined by replacing variables of a higher-level type with families of definitions of lower-level types. State representation is just as effective with lower-level types as with higher-level types, provided that the correct dependency relationships between lower-level data are introduced, and that the pattern of redefinition respects these relationships.

There are many examples of dependency maintenance at different levels. For instance:

- A model of a table can be considered as a point set at a low-level and as a family of features at a higher-level.
- The case-study of modelling a vehicle cruise control system [BBY92] includes

a model for a speed transducer. At a low-level of abstraction, observed dependency for the mechanics of the speed transducer are represented. At some higher-level, the model represents the speed transducer from an agent-oriented perspective.

- A DoNaLD high-level script is translated into an EDEN low-level script. It is possible to corrupt the dependencies expressed by DoNaLD definitions by interacting with them directly in EDEN.

In this chapter, issues relating to the implementation of definitive scripts are studied via one particular strategy of refinement of high-level scripts. This refinement takes a script in an existing high-level notation and transforms it into families of definitions in a low-level definitive script where there is only one data type: integer. These refined scripts can be represented in the framework of the *Dependency Maintainer Model* (DM Model). Mathematical sets and mappings are used to construct the model and to formulate the representation of definitive scripts. Translation to the DM Model establishes a bridge between observables that represent external semantics of a model (including its current value and dependencies) and variables that concern the internal semantics of representing a value on a computer.

A variable can be viewed as equivalent to an observable with no constraint upon its redefinition. The *block redefinition algorithm* can be used to implement protocols for redefinition that are consistent with redefinition of observables in a definitive script. Figure 4.1 shows the relationship between a high level-script of definitions, an integer-only low-level script of definitions with equivalent semantics (S) and a representation of this script by the DM Model ($\mathcal{M}_S$). It is possible to transform a high-level script to the lower-level representation but it is not necessarily possible to construct the high-level script from the low-level script. A DM Model $\mathcal{M}_S$ can represent the low-level script S using sets and mappings in such a way that

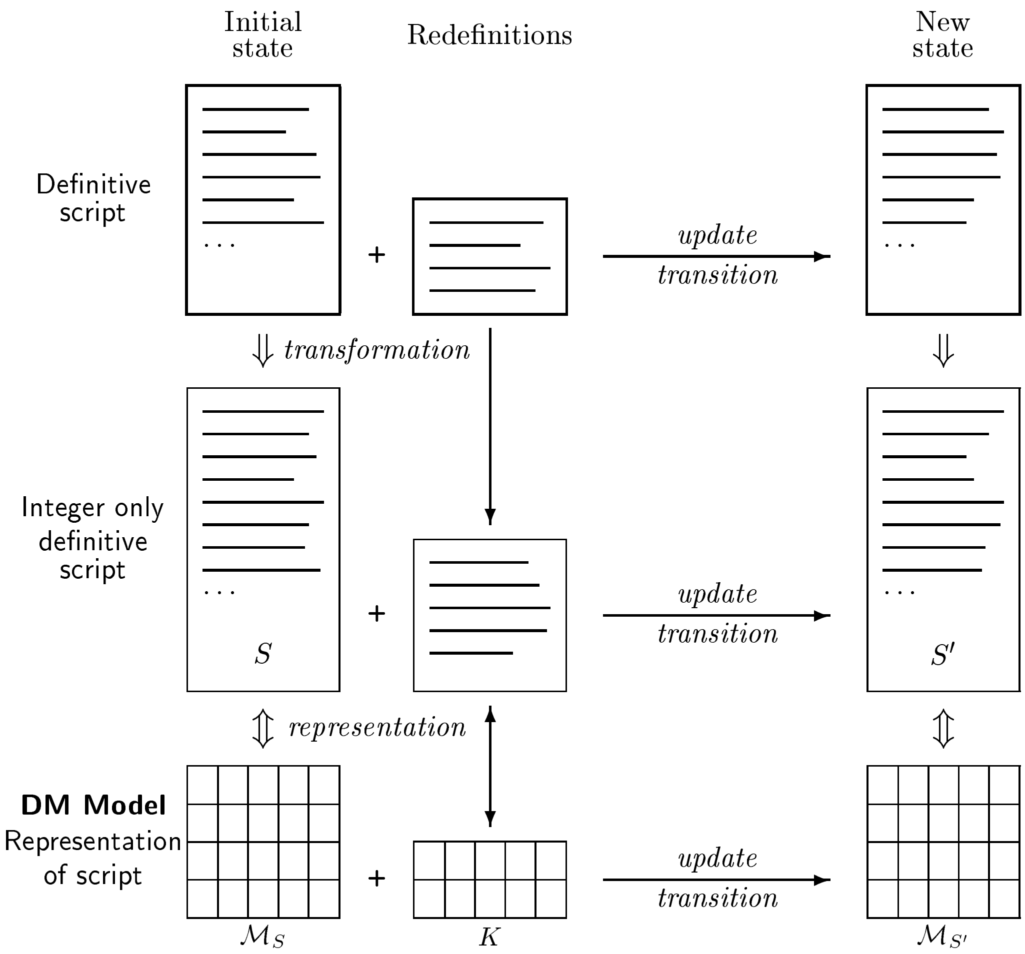


Figure 4.1: Definitive scripts and the DM Model.

the script can be reconstructed from the model. A low-level script such as S is said to be *DM Model representable.*

Redefinition of a high-level script induces an update state transition for the script. The new state of a high-level script can be transformed to the low-level script S', as can the block of redefinitions that resulted in the state transition. As illustrated in Figure 4.1, the low-level block of redefinitions can also be represented by the DM Model. The new state of the low-level script can be represented by a DM Model $\mathcal{M}_{S'}$. The block redefinition algorithm updates the DM Model consistently with the redefinitions to represent the new state. The current value associated with each observable in the low-level script is updated. Updating is the process by which we compute a new value consistent with redefinition.

In this chapter, the transformation process from high-level to low-level script is described. This is followed by an explanation of the representation of low-level scripts as DM Models. This model is used to formalise some important qualities of definitive scripts. These are used as a basis for discussion of strategies for update transition. The block redefinition algorithm is presented with an example of one update state transition. Graphical representations of *dependency structures* are introduced and case-studies use these to illustrate issues relating computational efficiency with dependency maintenance.

4.2 The DM Model for Definitive Scripts

In Chapter 3, the relationship between data structure and dependency is explored. The EDEN notation implements `boolean`, `integer`, `real`, `string` and `list` data types [YY88]. All models written in EDEN must use these types of data to represent their observation specific data. The `list` can often suffice for this purpose although there are some problems with the manipulation of data in lists.

Definitive script	First stage	Value
`a is (b + c) * d`	$a = times(add(b,c),d)$	$a = 48$
`b is power(e, 3)`	$b = power(e,3)$	$b = 8$
`c = 4`	$c = 4$	$c = 4$
`d is 2 * e`	$d = double(e)$	$d = 4$
`e = 2`	$e = 2$	$e = 2$

Table 4.1: A typical definitive script and stage 1 of script transformation.

There are many mechanisms in EDEN to handle functions, triggered actions and dependency maintenance, as described in Section 1.2.1. The motivation for building the DM Model is to strip away the overheads associated with dynamic interpretation of data types and functions, as well as the ordering of triggered procedural actions. Instead, the DM Model concentrates on the fundamental task of dependency maintenance. The only data type considered in the DM Model is the integers $\mathcal{Z}$. In the transformation from high-level scripts to low-level DM Model representable scripts, it is assumed that it is always possible to choose a large enough integer to represent the data types of the high-level script.

4.2.1 Transformation of Scripts

The DM Model is constructed by examining scripts of dependencies and extracting from these some common abstractions that can be used to reason about them. Every definition in a definitive script has an identity (a name) that is associated with both a value and a definition. For the purpose of the construction of a DM Model, it is assumed that the only data type is the integers $\mathcal{Z}$.

Table 4.1 shows an ideal definitive script for illustrating the transformation process, next to a script that represents the first stage of this process. The process of transforming any script to a script that can be represented by the DM Model has four stages. The values that the identities are associated with, consistent with their definition, are shown in the *value* column. From a script of definitions, it is

Reference	Second level of abstraction	Value
α	$a = times(x, d)$	$a = 48$
$\alpha + 1$	$b = power(e, y)$	$b = 8$
$\alpha + 2$	$c = 4$	$c = 4$
$\alpha + 3$	$d = double(e)$	$d = 4$
$\alpha + 4$	$e = 2$	$e = 2$
$\alpha + 5$	$x = add(b, c)$	$x = 12$
$\alpha + 6$	$y = 3$	$y = 3$

Table 4.2: Stage 2 of script transformation.

possible to calculate the value associated with an identifier from its definition and the definition of its dependencies. The DM Model represents both the value and the definition of an identifier simultaneously. The four stages for transforming a script into one that can be represented by the DM Model are listed below.

Stage 1 In a low-level script, every function in the definitions in the high-level script is replaced by a function that is a member of the set $\mathcal{F}$, where

$$\mathcal{F} = \{f \mid f : \mathcal{Z}^* \rightarrow \mathcal{Z}\} \tag{4.1}$$

The set $\mathcal{F}$ contains all functions that are mappings from sequences of integers to one integer value. This ensures that all arguments for functions and the resulting domain of these mappings are closed on the set of integers and that the DM Model has only one data type.

The script shown in the example in Table 4.1 requires very little transformation as it already represents dependency between integer values. The choice of how to represent values in the high-level script as integers is unimportant unless it is necessary for it to be possible to reconstruct the high-level script from the integer-only low-level script.

Stage 2 The next stage of transformation is to expand all function composition. In addition, explicit arguments to functions are replaced by adding new identifiers to the low-level script. An example of stage 2 for the scripts in Table 4.1 is shown in Table 4.2. The definition $b = power(e, 3)$ is replaced by $b = power(e, y)$, where $y = 3$ is a new definition in the low-level script.

Consider the functions g_1 and g_2 that are composed in a script to define the value of observable p to depend on observable q through the definition $p = g_1(g_2(q))$. This example definition is transformed into two definitions $p = g_1(r)$ and $r = g_2(q)$ in a low-level script, where r is chosen as a new identifier in the script.

The way in which an expression such as "`a is (b + c) / d`" is broken down into composed functions and new definitions is not considered in detail here. (For a particular illustration of this process, see Section 5.4.1.) An alternative method for expanding function composition without increasing the number of definitions is to use the function $expr_X$. This function represents the evaluation of a template expression X. For the example definition "`a is (b + c) / d`", the second stage of the transformation of `a` to a definition in a DM Model representable script is

$$a = expr_{(s_1+s_2)/s_3}(b, c, d) \tag{4.2}$$

The definition in equation 4.2 defines the value of a to be equal to $\frac{b+c}{d}$, with the associations to the template expression $s_1 = b$, $s_2 = c$, $s_3 = d$.

Stage 3 The third stage of transformation is to choose a range of integers between α and $\beta - 1$ that are to be used as unique integer *reference numbers* for each definition. The size of the range $\beta - \alpha$ is equal to the number of definitions after the second stage of transformation of the script. Table 4.2 shows the script from Table 4.1 with a choice of such suitable reference numbers following the expansion of all composed functions. Note that in this example, the values

Transformed Script	Reference $\in [\alpha, \beta - 1]$	Function $\in \mathcal{F}$	Dependencies $\in [\alpha, \beta - 1]^*$	Value $\in \mathcal{Z}$
$a = times(x, d)$	α	*times*	$(\alpha + 5, \alpha + 3)$	48
$b = power(e, y)$	$\alpha + 1$	*power*	$(\alpha + 4, \alpha + 6)$	8
$c = value_4()$	$\alpha + 2$	$value_4$	$()$	4
$d = double(e)$	$\alpha + 3$	*double*	$(\alpha + 4)$	4
$e = value_2()$	$\alpha + 4$	$value_2$	$()$	2
$x = add(b, c)$	$\alpha + 5$	*add*	$(\alpha + 1, \alpha + 2)$	12
$y = value_3()$	$\alpha + 6$	$value_3$	$()$	3

Table 4.3: Script transformed ready for representation by the DM model.

associated with the identifiers are consistent with the values defined by the high-level script, where the identities are common between the original script and its transformation.

Stage 4 The final stage of transformation to a DM Model representable script is to comply with a DM Model rule that every identifier must have a definition by a function. To achieve this, the generic function $value_i \in \mathcal{F}$ can be used, where for all integer values i and for all sequences of integers S the *value* function maps to i. This can be expressed by the proposition[1]

$$\forall i \in \mathcal{Z}, \forall S \in \mathcal{Z}^* \bullet value_i(S) = i \tag{4.3}$$

A transformed script can be represented by the DM Model. Table 4.3 shows the transformed example script prepared for this representation. The *function* column shows the name of the function used in the definition introduced in stage 1 and 2. The *reference* column shows the reference number in the range $[\alpha, \beta - 1]$ associated with each definition, in stage 3 of the transformation. If all the sequences of arguments to the functions that are part of the definitions have their elements

[1]The bullet symbol "•" is used in this chapter to represent *such that* in propositional statement.

substituted by their associated reference numbers, the sequences become as shown in the *dependencies* column.

4.2.2 Representing Definitive Scripts - The DM Model

Every reference number in the range $[\alpha, \beta - 1]$ can be mapped to a triple comprising:

- its associated value in $\mathcal{Z}$;
- its defining function in $\mathcal{F}$;
- a sequence of arguments to that function in $\mathcal{A}^*$.

This mapping is illustrated for the running example in the columns of Table 4.3. All identifiers are replaced by their reference number in all parts of the DM Model.

A transformed script S can be represented by a DM Model $\mathcal{M}_S$ that is given by a 4-tuple $(A, F, D, V) = \mathcal{M}_S$. Each element of the tuple is defined as shown below:

A - The range of reference numbers for the script S, where $A = [\alpha, \beta - 1]$ and the number of definitions in the DM Model is equal to $\beta - \alpha$.

F - A mapping $F : A \rightarrow \mathcal{F}$. Every reference number for a definition in the transformed script is mapped to its associated function by mapping F.

D - A mapping $D : A \rightarrow A^*$. Every reference number for a definition in the transformed script is mapped to its associated sequence of arguments by mapping D. The elements of the sequence are reference numbers, as shown in Table 4.3.

V - A mapping $V : A \rightarrow \mathcal{Z}$. Every reference number for a definition in the transformed script is mapped to the current (integer) value associated with that definition by mapping V.

For the example in Table 4.3, the DM model (A, F, D, V) would be represented by the following sets:

$$\begin{aligned}
A &= [\alpha, \alpha+6] \\
F &= \{\alpha \mapsto \mathit{times}, \alpha+1 \mapsto \mathit{power}, \alpha+2 \mapsto \mathit{value}_4, \alpha+3 \mapsto \mathit{double}, \\
&\quad \alpha+4 \mapsto \mathit{value}_2, \alpha+5 \mapsto \mathit{add}, \alpha+6 \mapsto \mathit{value}_3\} \\
D &= \{\alpha \mapsto (\alpha+5, \alpha+3), \alpha+1 \mapsto (\alpha+4, \alpha+6), \alpha+2 \mapsto (), \\
&\quad \alpha+3 \mapsto (\alpha+4), \alpha+4 \mapsto (), \alpha+5 \mapsto (\alpha+1, \alpha+2), \\
&\quad \alpha+6 \mapsto ()\} \\
V &= \{\alpha \mapsto 48, \alpha+1 \mapsto 8, \alpha+2 \mapsto 4, \alpha+3 \mapsto 4, \alpha+4 \mapsto 2, \\
&\quad \alpha+5 \mapsto 12, \alpha+6 \mapsto 3\}
\end{aligned}$$

An up-to-date model $\mathcal{M}_S$ is one in which every value is consistent with its definition. To formally define the concept of *up_to_date*, it is first necessary to define the function *lookup*. This maps a sequence of references to a sequence containing the values associated with to these references by DM Model, as given by mapping V. The sequence of values has the same order as the sequence of associated references, where

$$\mathit{lookup}_V : A^* \to \mathcal{Z}^*$$

$$\mathit{lookup}_V(a_0, \ldots, a_m) = (V(a_0), \ldots, V(a_m))$$

The boolean condition *up_to_date* — that is true when a script is up-to-date — can now be defined. The value associated with V for every reference number $a \in A$ in DM Model $\mathcal{M}_S$ is given by the evaluation of the associated function

$F(a)$. The arguments to this function are the sequence of *looked up* values for the dependencies of a, with references that are in the sequence $D(a)$. The condition can be expressed as

$$up\_to\_date(\mathcal{M}_S) \Leftrightarrow$$
$$\forall a \in A \bullet V(a) = F(a)(lookup_V(D(a)))$$

There is a problem determining the up-to-date status of a script if the sequence $D(a)$ contains a, where $D(a) = (a_0, \ldots, a_m) \wedge a \in \{a_0, \ldots, a_m\}$. In this situation, the process of updating $V(a)$ itself invokes an update of $V(a)$. For this to be consistent with the intended semantics of a definitive script, either the definition of $V(a)$ must be deemed invalid or the value of $V(a)$ undefined. In the context of the DM Model, the former convention is adopted.

Note that in the context of a parametric or variational modeller [SM95], a definition is interpreted as a simple form of equational constraint. This means that the equation

$$V(a) = F(a)((V(a_0), \ldots, V(a), \ldots, V(a_m))) \tag{4.4}$$

is acceptable provided that it can be solved for $V(a)$.

A definitive script that contains a reference that directly or indirectly depends on itself is said to have *cyclic dependency.* The next section examines the problem of identifying cyclic dependency in more detail.

4.2.3 Ordering in and the Status of DM Models

Given a DM Model representing a script, it is possible to determine whether the script contains any cyclic dependencies. When there is no cyclic dependency, the references in a DM Model can be partially ordered. This ordering can also be used to analyse the effect that structures in a model have on the efficiency of updating a script.

For a DM Model $\mathcal{M}_S$, it is possible to formalise the concept of dependency using the relation "$\lhd_D$" on set A. If $x \lhd_D y$ then the value of y is said to be *directly dependent* on the value of x. For two references x and y in A, the relation satisfies the condition

$$x \lhd_D y \Leftrightarrow D(y) = (a_0, \ldots, a_m) \wedge x \in \{a_0, \ldots, a_m\} \tag{4.5}$$

This relation $\lhd_D$ is not necessarily transitive. (A transitive relation $\prec$ over the integers is one for which $\forall x, y, z \in \mathcal{Z} \bullet (x \prec y) \wedge (y \prec z) \Rightarrow (x \prec z)$.) This is shown by the following counterexample. The script $a = 3, b = double(a), c = double(b)$ has DM Model representation with set $D = \{a \mapsto (), b \mapsto (a), c \mapsto (b)\}$. The relations for this model include $a \lhd_D b$ and $b \lhd_D c$. However, there is no relation $a \lhd_D c$ and so there is a script for which "$\lhd_D$" is not transitive.

The relation $\lhd_D$ can be used to define the set of all dependencies of a particular reference a in a model, as well as the set of all dependents for a reference. The direct dependencies of a are the elements of the sequence $D(a)$. The direct dependents of a are the references that have a as a dependency.

For any reference $a \in A$, the set of all references upon which the value of a directly depends — its *direct dependencies* — is given by the function *d_dependencies*, where[2]

$$d\_dependencies : A \rightarrow \mathcal{P}(A)$$

$$d\_dependencies(a) = \{x \mid x \lhd_D a\}$$

The set of all references that the value a depends on directly or indirectly is given by the function *dependencies*, where

$dependencies : A \rightarrow \mathcal{P}(A)$

[2] $\mathcal{P}(Z)$ represents the power-set (set of all subsets) of the integers $\mathcal{Z}$.

$$dependencies(a) = d\_dependencies(a) \cup \left(\bigcup_{x \in d\_dependencies(a)} dependencies(x) \right)$$

A very similar construction leads to the formal definition of dependents, references whose associated values should be updated if the value associated with a reference a is updated. The set given by function *d_dependents* contains all the references that have a as a direct dependency, where

$$d\_dependents : A \to \mathcal{P}(A)$$
$$d\_dependents(a) = \{x \mid a \lhd_D x\}$$

Elements of this set are known as *direct dependents* of a. Members of this set may in turn have additional dependents to the members of *d_dependents* and are *indirect dependents* of a. All the direct and indirect dependents of a are defined as elements of the set given by the function *dependents*, where

$$dependents : A \to \mathcal{P}(A)$$
$$dependents(a) = d\_dependents(a) \cup \left(\bigcup_{x \in d\_dependents(a)} dependents(x) \right)$$

A DM Model contains a cyclic dependency if there is a reference in the model that is directly or indirectly dependent upon itself. There are two ways to express the boolean condition *cyclic* for a particular DM Model $\mathcal{M}_S$, viz.

$$cylic(\mathcal{M}_S) \quad \Leftrightarrow \quad \exists a \in A \bullet a \in dependencies(a) \tag{4.6}$$

$$cylic(\mathcal{M}_S) \quad \Leftrightarrow \quad \exists a \in A \bullet a \in dependents(a) \tag{4.7}$$

A DM Model is in a *stable* state if all values associated with its references are consistent with their definition (up-to-date) and the model contains no cyclic dependencies. The boolean condition *stable* represents this conjunction, where

$$stable(\mathcal{M}_S) \Leftrightarrow up\_to\_date(\mathcal{M}_S) \land \neg cyclic(\mathcal{M}_S) \tag{4.8}$$

Block of redefinitions	DM Model representation		
	Reference	Function	Dependencies
`a is b * q`	α	*times*	$(\alpha+1, \alpha+7)$
`e = 1`	$\alpha+4$	$value_1$	$()$
`q = c div d`	$\alpha+7$	*div*	$(\alpha+2, \alpha+3)$

Table 4.4: A script of redefinitions and their DM Model representation.

In the context of figure 4.1, it is the stable states of the DM Model that can be interpreted at the higher levels of abstraction. The current status of a DM Model is consistent with the current status of the scripts it represents, both the original script and DM Model representable script. If the DM Model representing a script is *unstable* (not stable), then the script contains at least one cyclic dependency or the values have yet to be successfully updated. To be consistent with high-level interpretation, the state transition effected by an update algorithm should lead to a new stable state for a DM Model.

4.2.4 DM Model State Transitions

The *protected_update* procedure defined in this section performs a single *state transition* (or *update*) of the DM Model, from the current state $\mathcal{M}_S = (A, F, D, V)$ to a new state $\mathcal{M}_{S'} = (A', F', D', V')$. State transitions of the model represent the redefinitions of values and definitions in a transformed definitive script. It is assumed here that any redefinitions that update the DM Model are chosen appropriately to suit the underlying semantics of the transformed and high-level scripts (cf Figure 4.1).

One possible block of redefinitions for the script in Table 4.1 is shown in Table 4.4. Three redefinitions are shown in the left hand column with a DM Model representation of these redefinitions in the right-hand three columns. The process of transforming a block of redefinitions from a general definitive script to a DM Model representable block of redefinitions is the same as the four stage transformation of a script of definitions. If the same identifier exists in the script of definitions and the block of redefinitions, then it must be chosen to have the same reference number in stage 3.

In a high-level definitive script, the definition associated with an identifier is altered if it exists as a redefinition in a state transition. In a similar way, a state transition from a DM Model $\mathcal{M}_S$ alters the mappings associated with a reference that appears in the block of redefinitions. A block of redefinitions can associate new dependencies with existing references and introduce completely new references into a DM Model. Block redefinition can also make an existing reference depend on a newly introduced reference. For example, after the transition with the redefinition of a in the running example, a dependency on the new identity q with new reference number $\alpha + 7$ is introduced.

For any state of a DM Model, an update can only occur when there is a set of redefinitions K. Each element of this set is a mapping from a reference in a new range of references $A' = [\alpha', \beta' - 1]$ to its new definition. The range A' is an extension of the range A. In general, it is possible to extend this range to accommodate any finite number of redefinitions in a block of redefinitions. This extension occurs when references associated with new identifiers for a script of definitions exist in a block of redefinitions. This is illustrated in the example by the introduction of the definition "q". It is possible to add to the number of definitions represented in a DM Model through update transitions. It is not possible to remove represented definitions in this way.

A block of redefinitions for a script S with DM Model $\mathcal{M}_S$ and known new range A' known can be represented by a mapping K, where

$$K \subset \{f \mid f : A' \to \mathcal{F} \times A'^*\} \tag{4.9}$$

In the mapping K, each reference number in the block of redefinitions is mapped to a pair comprising the function part of the redefinition followed by the references for the sequence of arguments (dependencies) for this function. By way of illustration, the block of redefinitions shown in Table 4.4 is represented by the following mapping K:

$$\begin{aligned} K \;\; = \;\; & \{\alpha \mapsto (\mathit{times}, (\alpha+1, \alpha+7)), \\ & \alpha+4 \mapsto (\mathit{value}_1, ()) \\ & \alpha+7 \mapsto (\mathit{div}, (\alpha+2, \alpha+3))\} \end{aligned}$$

K_F and K_D will be used to denote the projections of the map K onto the function and domain components of its range respectively.

$$K_F = \{a \mapsto f \mid \exists S \in A'^* \bullet (a \mapsto (f, S)) \in K\} \tag{4.10}$$

$$K_D = \{a \mapsto S \mid \exists f \in \mathcal{F} \bullet (a \mapsto (f, S)) \in K\} \tag{4.11}$$

For the purposes of describing the update, it is assumed that the references in K are always well chosen to be in a range $A' = [\alpha', \beta' - 1] \supseteq [\alpha, \beta - 1] = A$. The set K of redefinitions for a DM Model is *suitable* provided that it contains a new definitions for each additional new reference:

$$\begin{aligned} & \mathit{suitable}(K, A, A') \Leftrightarrow \\ & \qquad \forall a \in (A' \setminus A), \exists f \in \mathcal{F}, \exists S \in A'^* \bullet (a \mapsto (f, S)) \in K \end{aligned}$$

To define the functions that specify the update process, the "$\oplus$" symbol is used in the same way as in the Z notation [Spi92]. For two relations (mappings) with the same domain and range Q and R, the relation $Q \oplus R$ relates everything in the domain of R to the same objects as R does, and everything else in the domain of Q to the same objects as Q does. The operator dom is introduced to map relations to the sets of objects they reference. For any relation Q, "dom Q" is a set containing all the objects in the domain of Q.

The update of F to F', the reference number to associated defining function mapping of the DM Model, is described by the function *upf*, where

$$\begin{aligned} \mathit{upf} &: \{g \mid g : \mathcal{Z} \to \mathcal{F}\} \times \{g \mid g : \mathcal{Z} \to \mathcal{F}\} \\ &\quad \to \{g \mid g : \mathcal{Z} \to \mathcal{F}\} \\ \mathit{upf}(F, K_F) &= F \oplus K_F \end{aligned}$$

Any references common to F and K are removed from F and then these are combined with the new association of functions with references in K. This specification of the update process demonstrates one of the strengths of a DM Model. The set of all possible functions used in the model does not have to be predetermined prior to its initial state.

Similarly, the function *upd* specifies the state transition from D to D', where

$$\begin{aligned} \mathit{upd} &: \{g \mid g : \mathcal{Z} \to \mathcal{Z}^*\} \times \{g \mid g : \mathcal{Z} \to \mathcal{Z}^*\} \\ &\quad \to \{g \mid g : \mathcal{Z} \to \mathcal{Z}^*\} \\ \mathit{upd}(D, K_D) &= D \oplus K_D \end{aligned}$$

This function updates the dependencies associated with a reference to be consistent with the redefinitions in set K. Each of the new redefinitions could potentially introduce cyclic dependency into the new state of the model. As an example of this problem, consider the redefinition represented by $K = \{\alpha \mapsto (\mathit{double}, (\alpha))\}$, where the update of a DM Model would result in a reference having a direct dependence on itself.

The new set of values V' associated to references is calculated from the updated functions and dependencies and is not directly related to the set of redefinitions K. After the update transition the DM Model should be up-to-date, where every value is consistent with its definition in the new state. Following the convention introduced in Section 4.2.2, if the new state of D' introduces cyclic dependency into the model then the update must be deemed invalid. In the absence of cyclic dependency, the *up_to_date* condition should hold for $\mathcal{M}_{S'}$:

$$\forall a \in A' \bullet V'(a) = F'(a)(lookup_{V'}(D'(a))) \tag{4.12}$$

To achieve this, it is necessary to update the values all references associated with redefinitions and their dependents. For a given block of redefinitions K, the set of references whose value is updated is $U \equiv U(K)$, where

$$U(K) = \mathrm{dom}\ K \cup \left(\bigcup_{x \in \mathrm{dom}\ K} dependents(x) \right) \tag{4.13}$$

A protected update is one where for any given block of redefinitions K, the new state of the model following an update is still a stable state for a DM Model, even if this means that the block of redefinitions is ignored. To achieve this, the *protected_update* mapping from a DM Model and a block of redefinitions to a new state of the DM Model either has a suitable set of redefinitions that take it to a stable state or the redefinitions are rejected and the state transition does not affect the DM Model. This protection is formalised by the mapping *protected_update*, where $F' = upf(F, K_F)$, $D' = upd(D, K_D)$ and

$$protected\_update(\mathcal{M}_S, K) = \begin{cases} \mathcal{M}_S & \begin{array}{c}\text{if } cyclic((A', F', D', V)) \\ \text{or } \neg suitable(K)\end{array} \\ (A', F', D', V') & \text{otherwise} \end{cases} \tag{4.14}$$

The values of all the references in U should be updated once K is shown to be suitable and the potential introduction of cyclic dependency by the block of redefinitions has been ruled out. The most efficient ordering in which to update the values associated with the references in U for the condition $up\_to\_date(\mathcal{M}_{S'})$ to be true is discussed in Section 4.3.2. For a block of redefinitions K, the update of a DM Model $\mathcal{M}_S$ to its new state $\mathcal{M}_{S'}$, which is consistent with the update of the

Script Represented	Reference $\in [\alpha', \beta' - 1] = A'$	Function $\in \mathcal{F}$	Dependencies $\in A'^*$	Value $\in \mathcal{Z}$
$a = times(d, q)$	α	$times$	$(\alpha+1, \alpha+7)$	4
$b = power(e, y)$	$\alpha+1$	$power$	$(\alpha+4, \alpha+6)$	1
$c = value_4()$	$\alpha+2$	$value_4$	$()$	4
$d = double(e)$	$\alpha+3$	$double$	$(\alpha+4)$	2
$e = value_1()$	$\alpha+4$	$value_1$	$()$	1
$x = add(b, c)$	$\alpha+5$	add	$(\alpha+1, \alpha+2)$	5
$y = value_3()$	$\alpha+6$	$value_3$	$()$	3
$q = div(c, d)$	$\alpha+7$	div	$(\alpha+2, \alpha+3)$	2

Table 4.5: Example of an updated DM Model.

definitive script S that it represents to a new state S', is

$$\mathcal{M}_{S'} = protected\_update(\mathcal{M}_S, K) \tag{4.15}$$

An example of the state of the DM Model after an update is shown in Table 4.5. This example is based on the running example through this chapter (the transformed script in Table 4.3 and the block of redefinitions in Table 4.4). The definition of a has been altered, breaking its dependency to x (reference number $\alpha+5$) and adding a dependency to the new definition q (reference number $\alpha+7$). The definition of e has also changed so that its value is defined by a different *value* function. A new definition has been added so that the range A' is now $[\alpha', \beta'-1] = [\alpha, \alpha+7]$. Six out of the nine values are updated as a result of the state transition, and $\mathcal{M}_{S'}$ is a stable model.

4.2.5 Incremental Construction of DM Models

The DM Model is constructed by the transformation and representation of scripts that already exist. It is possible to construct a DM Model from scratch in the same way that a definitive script can be incrementally constructed from scratch. In this case, the initial state for any DM Model is represented by $\mathcal{M}_\emptyset = (\emptyset, \emptyset, \emptyset, \emptyset)$, a 4-tuple containing four empty sets $\emptyset$. Movement to the next state $\mathcal{M}_S$ is achieved in the same way as any state transition for the DM Model, through the introduction of a block of redefinitions K that is a representation of the first script S for the new DM Model. This initialization is possible by application of an algorithm such as the block redefinition algorithm presented in Section 4.3.3.

There are no preconceptions about the script, the structure of the script, or its DM Model representation prior to the incremental construction of a model. The initial reference range, functions, sequences of dependencies and values do not have to be predetermined before the initial script of (re)definitions K is known. The process of initialising a DM Model is the same as the process of updating it through state transition. Note that this is consistent with the operation of the existing *Tkeden* interpreter.

4.2.6 Interaction Machines

Wegner introduces the concept of *interaction machines* in [Weg97] and claims that these are more powerful than rule-based algorithms. These are defined as Turing machines extended by the addition of input and output actions that support dynamic interaction with an external environment. The DM Model can be viewed as an interaction machine, with input in the form of redefinitions and output in the form of current values and script represented after update state transitions. A script can be changed beyond recognition by a suitable block of redefinitions. A rule-based algorithm running on a Turing machine cannot be modified until its execution has terminated.

Wegner argues for the radical notion that interactive systems are more powerful problem-solving engines than algorithms and proposes a new paradigm for computing based around the unifying concept of interaction. He notes that interfaces to open systems often constrain interactive behaviour and restrict constituent interface components to goal-directed behaviour. This is because the construction of these interfaces is necessarily algorithmic using current programming paradigms. Wegner's vision is closely connected with the conflation of computer programming and computer use discussed in Chapter 1 1.1. This conflation of roles has been reasonably successfully demonstrated in the use of the EDEN interpreter (cf the case-study in Section 2.3.2). The work in this thesis is principally concerned with yet another conflation of roles whereby the user and the programmer can develop the computer as an instrument. This conflation is beyond the scope of EDEN.

4.3 Algorithms for DM Machine Update

The DM Model is defined by mappings over sets that represent a script of definitions. This section examines algorithms that both effect a state transition for a script and ensure that the values in the DM Model are up-to-date following the state transition. The new *block redefinition algorithm* for performing the *protected_update* function and transition from V to V' for any DM Model is presented.

The concept of a *dependency structure* is for a DM Model introduced and a graphical representation of these structures is described. This structure is used to explain informally the benefits of the block redefinition algorithm in comparison to previous strategies for the maintenance of dependencies. Dependency structure is also used to illustrate one state transition using this algorithm.

4.3.1 Dependency Structure

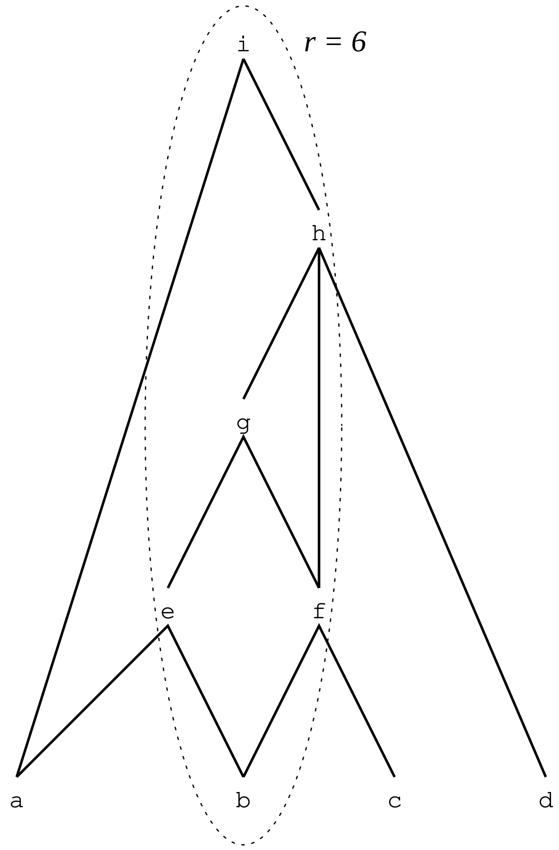
The relation "$\lhd_D$" on the set A for DM Model $\mathcal{M}_S$ defines a *dependency structure*. Dependency structures can be used to analyse patterns of dependency in represented scripts. It is possible to draw graphical representations of dependency structures that can be used as cognitive artefacts for the exploration of dependencies in a script. This can inform the choice of redefinitions or the construction of new definitive scripts to represent similar observed dependencies in the real world.

The script S represented in Figure 4.2 is used as a new running example to demonstrate the drawing of dependency structures and the application of the block redefinition algorithm. In this figure, the combinatorial graph depicts the *dependency structure* over references in a DM Model $\mathcal{M}_S$ of S. The relation $\lhd_D$ for $\mathcal{M}_S$ is represented on the diagram by edges between the vertices of the graph. An edge exists between vertices with associated reference numbers x and y in the graph if $x \lhd_D y$. If this is the case, vertex y is the parent of child node x. In this representation, the graph is directed and acyclic with directional arrows replaced by orientation up the page. The table in the figure depicts the example script S together with the dependency relations on the set A.

4.3.2 Comparing Update Strategies

In this section, the strategy for ordering the update of the values associated with references in the DM Model (the set V) is considered. The arguments presented are based on the premise that each update is expensive in comparison with the calculation of the update ordering. It could be the case that every data item mapped to by V has been chosen in transformation to be a large integer that represents a complex piece of geometry in a high-level definitive script for geometry. The computation of the functions in F may be computationally expensive for these representations. For example, if an integer represents arbitrary point sets, then an operator such as a blend of two such point sets and the rendering action that produces an image of the blend is likely to be a processor intensive task for a computer system. In such a context, there is a string motivation to determine the optimal number of updates required to effect a particular state transition.

The DM Model representing a transformed script is in a stable state $\mathcal{M}_S$ if all values of words in definitive store are consistent with their definition. During the



Script S	The relation $\lhd_D$ on S	
$a = 3$	$a \lhd_D e$	$a \lhd_D i$
$b = 10$	$b \lhd_D e$	$b \lhd_D f$
$c = 7$	$c \lhd_D f$	
$d = 9$	$d \lhd_D h$	
$e = add(a, b)$	$e \lhd_D g$	
$f = add(b, c)$	$f \lhd_D g$	$f \lhd_D h$
$g = times(e, f)$	$g \lhd_D h$	
$h = max_3(g, f, d)$	$h \lhd_D i$	
$i = power(h, a)$		

Figure 4.2: A Dependency Structure for S

state transition to state $\mathcal{M}_{S'}$ the values of all references contained in the domain of mapping K, considered as *redefinitions*, must be updated. All values associated with references in the set $U(K)$ corresponding to K must also be updated[3], as these values may have changed. One possible strategy to perform these calculations is to update the value of each reference repeatedly until it is observed that condition *up_to_date*$(\mathcal{M}_{S'})$ holds. This is inefficient as if there are $n = \parallel A' \parallel$ vertices in a dependency structure then this method requires $\mathcal{O}(n^2)$ updates to guarantee the DM Model reaches a stable state. For this update strategy, each value has to be calculated n times to guarantee consistency in the worst case.

Knuth [Knu68] presents an algorithm in for topological sorting of the elements of a partially ordered set. In the context of the DM Model $\mathcal{M}_S$, Knuth's algorithm determines an efficient ordering for updating the value of every reference in A'. This reduces the number of updates required to n, as, once the sort has taken place, each value only has to be updates once. However, it is often the case that the block of redefinitions K is smaller than the reference range A'. In this case, it may not be necessary to update all n values, as $\parallel U(K) \parallel \leq n$.

In the use of the existing *Tkeden* interpreter, the size of the sets of redefinition is typically of the order of one or two. The interpreter itself can only handle one redefinition at a time ($\parallel K \parallel = 1$) and is optimised for this. It often performs a large number of unnecessary calculations. Each Tkeden single redefinition initiates a state transition, even though several redefinitions (a block) presented to the interpreter at the same time may conceivably pertain to the same change of state in the observed model. A better strategy is to consider one state transition as defined by a block of one or more simultaneous redefinitions. In the context of Figure 4.1, a single redefinition in a high-level script may be transformed into a block of redefinitions at the low-level. This fact gives strong motivation for the block redefinition algorithm.

[3]Consider the propagation of change in a spreadsheet after changing the value in one cell.

The useful measure $r \equiv r_S$ can be associated with the dependency structure of a script S. The purpose of r_S is to describe an upper bound on the number of updates associated with a single redefinition in S. The measure r is formally defined as

$$r_S = \max_{a \in A}\{(\| \, dependents(a) \, \| + 1)\} \tag{4.16}$$

As a reference in a script can have at most $n - 1$ dependents, the value of r_S is bounded by and often significantly less than n. For the script in Figure 4.2, $r_S = 6$.

In *Tkeden*, the absence of block redefinition means that the redefinitions in K are processed one-by-one. This process results in $\| K \|$ state transitions. The number of associated recalculations of values in U for K has an upper bound of $r \| K \|$. For a dependency structure with a large value for r, it is possible for this upper bound to exceed the n updates required to update all the values of A'.

To determine an ordering for efficient update where only small subsets of the model are redefined requires the modification of an algorithm such as Knuth's topological sort. The algorithm for efficient update of the DM Model is called the *block redefinition* algorithm, where the word *block* signifies that the algorithm considers several redefinitions simultaneously. This algorithm finds a sensible ordering of work to be done, in the context of both the current dependency structure for the script represented all the redefinitions in the block of redefinitions. The upper bound for the number of updates is at worst n and the actual number of updates is often lower, depending on the context of the current dependency structure and the block of redefinitions.

By way of illustration, consider the example shown in Figure 4.2. A sensible order for updating the values of all references would follow the topological sort a, b, c, d, e, f, g, h and i. If the two redefinitions

$$d \;\; = \;\; 23$$

$$h \;=\; min(g, c, d)$$

are redefined in one state transition, it is only necessary to update (set the value of) d, update the new value of h and then recalculate i. The values of a, b, c, e, f and g all remain the same.

4.3.3 The Block Redefinition Algorithm

The block redefinition algorithm, developed by adapting the Knuth algorithm for topological sorting, has three distinct phases. For DM Model $\mathcal{M}_S = (A, F, D, V)$:

- the first phase updates D and F to D' and F' respectively, so that they are consistent with the redefinitions in K. There is no check for cyclic dependency in this phase.

- the second phase calculates the optimal order in which the values of V need to be updated so that the condition $up\_to\_date(\mathcal{M}_{S'})$ is true. During this phase, cyclic dependencies can be detected. If cyclicity is detected, the changes made in phase one are revoked and $\mathcal{M}_{S'}$ is set to be the same as $\mathcal{M}_S$, as for the *protected_update* function.

- the third phase performs the updates (step 6 of phase 3) in the order determined in the second phase.

In the specification of the algorithm, an additional mapping $kc : A' \rightarrow \mathcal{Z}$, complementary to those that define the DM Model, is used to introduce numerical counters for each reference. These counters are referred to as the *Knuth counters* as they resemble the counters used in Knuth's topological sort algorithm. In the exposition of the algorithm, it is assumed that, with an appropriate choice of data structures to represent the DM Model mappings, the calculation of the function *d_dependents* is trivial. A suitable data representation for this purpose is presented in the description of the DAM Machine implementation in Chapter 5.

The following definitions and notations are used in presenting the block redefinition algorithm:

- a function *seq* to construct an enumeration of a sequence. The sequence $seq(Q)$ is $(q_1, \ldots, q_{||Q||})$ and every element of the sequence is a unique member of the set Q.

- the standard list processing functions *head*, *tail*, *append*, *length* and *contains* on sequences. These functions are: *head* maps to the first element of a sequence, *tail* maps to a sequence with the head element removed, *append* adds a new element onto the end of a sequence, *contains* is a boolean function to test whether a sequence contains a particular element.

$$head((t_1, \ldots, t_n)) = t_1 \tag{4.17}$$

$$tail((t_1, \ldots, t_n)) = (t_2, \ldots, t_n) \tag{4.18}$$

$$append((t_1, \ldots, t_n), x) = (t_1, \ldots, t_n, x) \tag{4.19}$$

$$length((t_1, \ldots, t_n)) = n \tag{4.20}$$

$$contains((t_1, \ldots, t_n), x) \Leftrightarrow \exists k \in [1, n] \bullet x = t_k \tag{4.21}$$

The three phases of the block redefinition algorithm are described step-by-step below. The state of the model prior to execution is $\mathcal{M}_S = (A, F, D, V)$ and the block of redefinitions is K. The new range A' is known prior to algorithm execution. The new state of the DM Model after the algorithm execution for one state transition is $\mathcal{M}_{S'} = (A', F', D', V')$.

Phase 1 1. $kc := \{a \mapsto b \mid a \in A', b = 0\}$.

2. $B := seq(\text{dom } K)$.

3. for $j = 1$ to $length(B)$

    do

    $F := F \oplus \{B_j \mapsto K_F(B_j)\}$.

    $D := D \oplus \{B_j \mapsto K_D(B_j)\}$.

    done

4. Continue to step 1 of phase 2.

Phase 2 1. $a := head(B)$.

2. $C := seq(d\_dependents(a))$.

3. for $j = 1$ to $length(C)$

    do

    $kc := kc \oplus \{C_j \mapsto max(kc(a) + 1, kc(C_j))\}$

    if $\neg contains(B, C_j)$

    then $B := append(B, C_j)$

    done

4. $B := tail(B)$.

5. if $kc(a) \geq (\beta' - \alpha')$

    then report a *cyclic dependency*, set $\mathcal{M}_{S'} = \mathcal{M}_S$ and halt.

6. if $length(B) > 0$

    then return to step 1 of phase 2,

    else continue to step 1 of phase 3.

Phase 3 1. $B := seq(\text{dom } K)$.

2. for $j = 1$ to $length(B)$

    do (β' is used as a marker.)

    if $kc(B_j) \neq 0$ then $B_j := \beta'$.

done

3. $a := head(B)$.

4. If $a := \beta'$ then skip to step 8 of this phase.

5. $C := seq(d\_dependents(a))$.

6. $V := V \oplus \{a \mapsto F(a)(lookup_V(D(a)))\}$.

7. for $j = 1$ to $length(C)$

   do

   if $(kc(C_j) = (kc(a) + 1)) \wedge (\neg contains(B, C_j))$

   then $B := append(B, C_j)$

   done

8. $B := tail(B)$.

9. if $length(B) > 0$

   then go to step 3 of phase 3,

   else go to the end.

End $F' = F$, $D' = D$, $V' = V$ and $M_{S'} = (A', F', D', V')$.

The detection of cyclic dependency in phase 2 is simplistic[4]. At the end of phase 2, the values of the Knuth counters for each reference indicate the order in which the values should be updated, starting at 0. References that have the same Knuth counter after phase 2 can have their values safely updated in parallel.

If there is a cyclic dependency in the model, then the value of a Knuth Counter associated with a reference will eventually be incremented to a value greater than the total numbers of vertices in the dependency structure. This way in which this process operates in illustrated in Figure 4.3, where the Knuth counters in the

[4] i.e. it uses an indirect and relatively inefficient method to identify a simple cyclic dependency such as a self-reference.

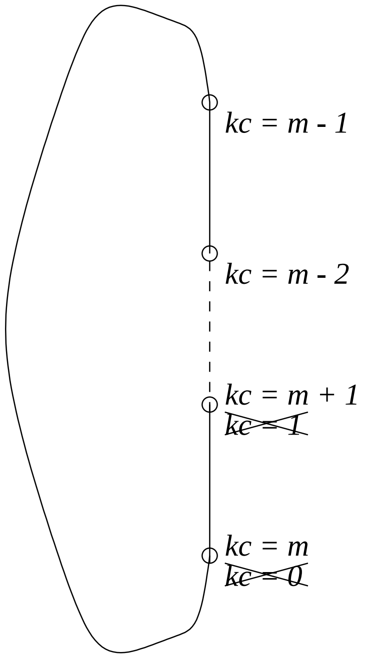


Figure 4.3: The numbering of a structure containing cyclic dependency.

loop of nodes increment repeatedly. In this case, the Knuth counter calculation in phase 2 of the block redefinition will terminate the algorithm because of the assignment of the value $\beta - \alpha$ to a counter.

4.3.4 Example of the Block Redefinition Algorithm

Figures 4.4, 4.5, 4.6 and Table 4.6 trace a sample execution of the block redefinition algorithm for one state transition that updates script S shown in Figure 4.2. The model corresponding to this structure, $\mathcal{M}_S = (A, F, D, V)$, is represented in Figure 4.4a, where the identifiers associated with the vertices in the dependency structure are shown in the left-hand-side of each split box representation and the current value of the Knuth counter associated with the identifier is shown on the right-hand-side. In this example, the sets that make up the DM Model 4-tuple representing the script are as follows[5]:

$$
\begin{aligned}
A &= [a, i] \\
F &= \{a \mapsto value_3, b \mapsto value_{10}, c \mapsto value_7, d \mapsto value_9, \\
&\quad e \mapsto add, f \mapsto add, g \mapsto times, h \mapsto max_3, i \mapsto power\} \\
D &= \{a \mapsto (), b \mapsto (), c \mapsto (), d \mapsto (), e \mapsto (a, b), \\
&\quad f \mapsto (b, c), g \mapsto (e, f), h \mapsto (g, f, d), i \mapsto (h, a)\} \\
V &= \{a \mapsto 3, b \mapsto 10, c \mapsto 7, d \mapsto 9, e \mapsto 13, f \mapsto 17, \\
&\quad g \mapsto 221, h \mapsto 221, i \mapsto 10793861\}
\end{aligned}
$$

The block of redefinitions to be introduced is

$$K = \{a \mapsto (value_2, ()), d \mapsto (double, (c)), g \mapsto (add, (e, f))\} \tag{4.22}$$

The execution of the algorithm will perform the transition of the DM Model from state $\mathcal{M}_S$ to $\mathcal{M}_{S'}$.

Figure 4.4 is used to explain phase 1 of the block redefinition algorithm and is split into three separate sections labeled **a**, **b** and **c**.

a - a graphical representation of the dependency structure of $\mathcal{M}_S$, with all Knuth counters initialised to zero.

b - At step 2 of phase 1, $B = seq(\text{dom } K)$ is the sequence (a, d, g). Vertices in the structure that are in the initial sequence B are highlighted with boxes that have thicker borders.

c - By the end of phase 1, D and F are updated. The redefinition of d in K has changed the topology of the dependency structure so that now $c \lhd_D d$. The values in V are no longer consistent with their definition ($\neg up\_to\_date(\mathcal{M}_S)$).

[5]Function definitions are as follows: $add(x, y) = x + y$, $times(x, y) = xy$, $max_3(x, y, z) = max(max(x, y), z)$, $power(x, y) = x^y$ and $double(x) = 2x$. For clarity of the exposition, identifiers are used in place of their associated reference numbers.

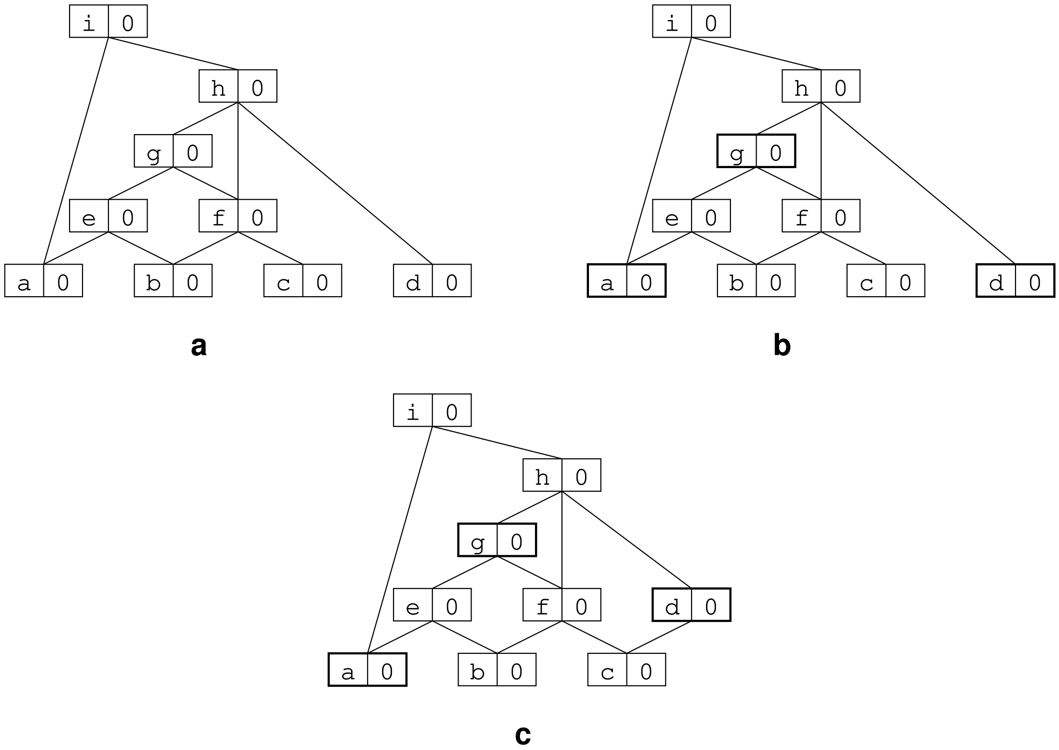


Figure 4.4: Phase 1 of the block redefinition algorithm.

It is possible that a cyclic dependency has been introduced into the structure during phase 1. This is detected in phase 2, which is represented in nine dependency structures across Figures 4.5 and 4.6. In the example, the set of redefinitions K does not introduce any cyclic dependency. The *initial* state for phase 2 is depicted in the top left-hand graph of Figure 4.5. Each graph in the figures is annotated by a value of n, which indicates that it represents the n^{th} iteration of phase 2 of the block redefinition algorithm. For each iteration, the graph representation is a snapshot after step 5 of the phase. The state of current values of sequences B and C are shown below each dependency structure. The current value for a in the algorithm, the head of the sequence B at the start of an iteration of the phase, is depicted by a circled node.

$n = 0$ - Vertex $head(B) = a$ is the first element in the sequence B. C is a sequence

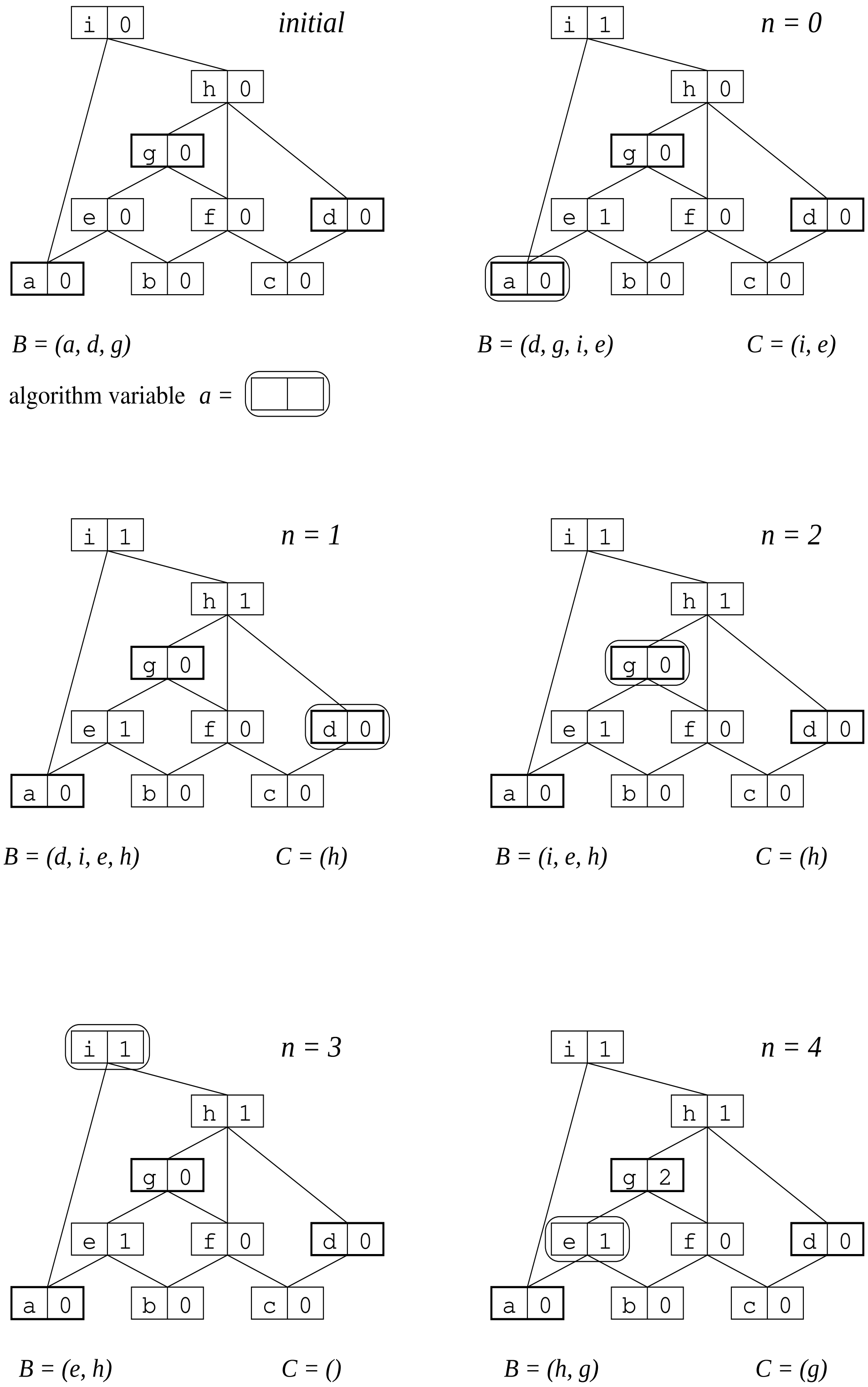


Figure 4.5: Intermediate states after n-th iteration in phase 2 of the block redefinition algorithm ($0 \leq n \leq 4$).

of the dependents of a, which in the example are i and e. Each of these have a Knuth Counter of 0 that is incremented in this first iteration of the phase to $kc(a) + 1 = 1$. As B does not yet contain i or e, these are appended to the end of the sequence. The head element is removed from sequence B before the next iteration.

$n = 1$ - Similar procedure to $n = 0$, except that $head(B) = d$ and $C = (h)$. The only direct dependent of d is node h that has its Knuth counter set to 1. As the sequence B does not contain h, it is appended to the end of B.

$n = 2$ - $head(B) = g$. The only dependent of g is h that already has $kc(h) = 1$. The new value of the Knuth counter for h is set to be $max(kc(h), kc(g) + 1) = max(1, 1) = 1$. Node h is already in sequence B and so is not appended to the end of this sequence during this iteration.

$n = 3$ - $head(B) = i$. There are no dependents for i so sequence $C = ()$, the empty sequence. Step 3 is not executed on this iteration and as a result and the algorithm proceeds to setting B to the sequence that is the tail of B removing the head element i from B.

$n = 4$ - $head(B) = e$. The value of the Knuth Counter for e is $kc(e) = 1$ and for its dependent g is $kc(g) = 0$. The new counter value for g is set to be $max(kc(e) + 1, kc(g)) = max(2, 1) = 2$. Element g is appended to sequence B and the head element e is removed.

The next five iterations of phase 2 of the algorithm are shown in Figure 4.6 for values $n = 5$ up to $n = 9$.

$n = 5$ - $head(B) = h$. This is the second time that h has been the node that is at the head of B in an iteration of the phase. The only node that is dependent

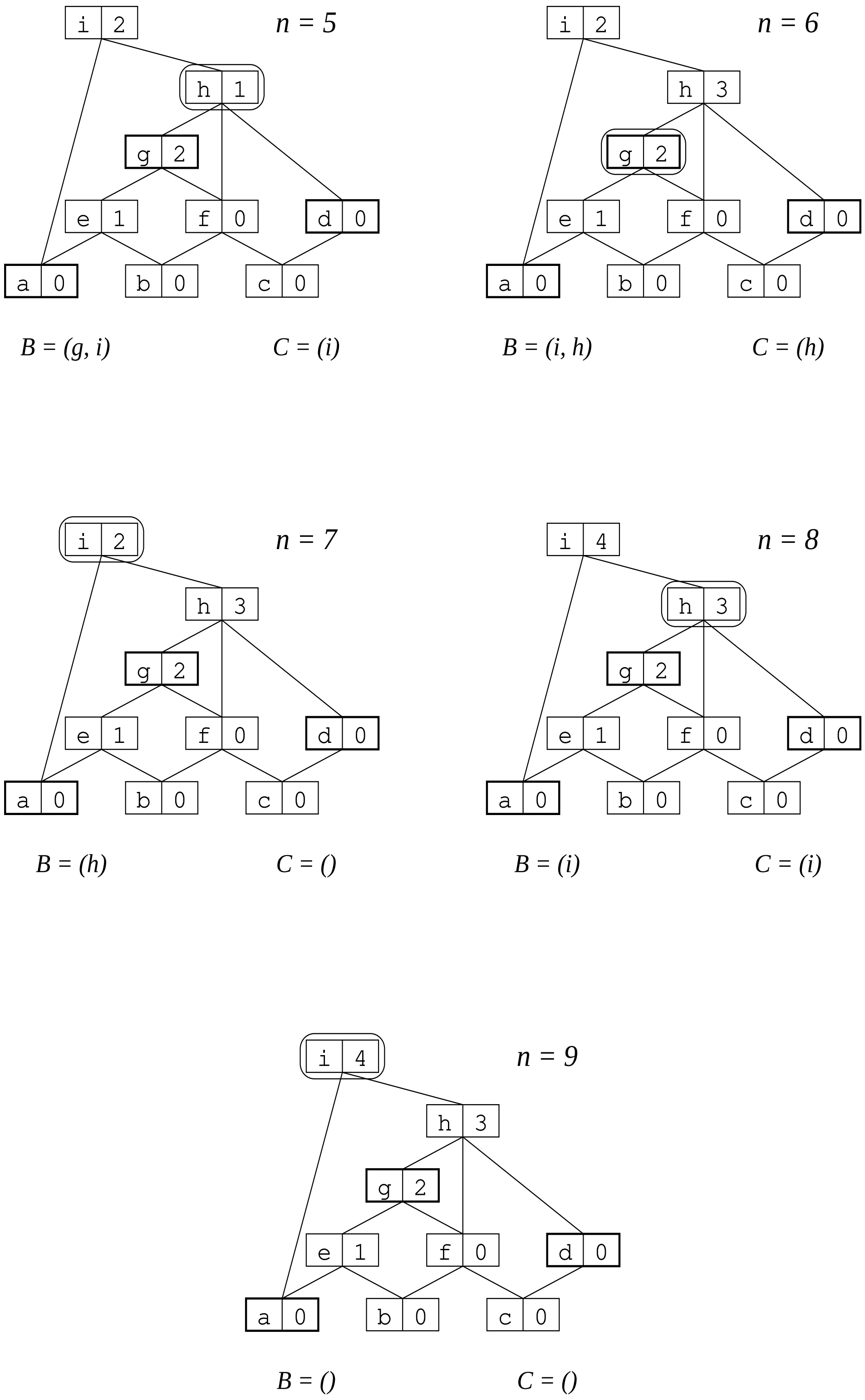


Figure 4.6: Intermediate states after n-th iteration in phase 2 of the block redefinition algorithm ($5 \leq n \leq 9$).

Iteration	Sequence B	Sequence C	Updates
initial	(a, d, g)		$\downarrow$ end of step 2
$n = 0$	(a, d, β')	(i, e)	$V(a) := value_2()$
$n = 1$	(d, β', e)	(h)	$V(d) := double(V(c))$
$n = 2$	(β', e)		—
$n = 3$	(e)	(g)	$V(e) := add(V(a), V(b))$
$n = 4$	(g)	(h)	$V(g) := add(V(e), V(f))$
$n = 5$	(h)	(i)	$V(h) := max_3(V(g), V(f), V(d))$
$n = 6$	(i)	$()$	$V(i) := power(V(h), V(a))$

Table 4.6: Phase 3 of the block redefinition algorithm.

on h is i and as $kc(h) = 1$ so the new counter for i is set to $kc(h) + 1 = 2$. B already contains element i.

$n = 6$ - $head(B) = g$. The Knuth counter for its direct dependent h is set to $kc(g) + 1 = 3$ and h is appended to sequence B.

$n = 7$ - $head(B) = i$ and as there are no dependents for i, step 3 is not executed and i is removed from B.

$n = 8$ - $head(B) = h$ for the third time in the phase. The counter for i is updated to be $kc(h) + 1 = 4$.

$n = 9$ - $head(B) = i$. As for $n = 7$ node i has no dependents so step 3 is not executed and i is removed from the head of B. The length of B is zero so the algorithm proceeds to phase 3.

At the end of phase 2, the topological sorting for the ordering of updates in complete. All nodes with a Knuth counter of zero and a thick border should be updated first, followed by any vertex with a Knuth counter of 1, then 2, and so on. This update is carried out in step 6 of phase 3.

Intermediate states during phase 3 are represented in Table 4.6. Each row in the table corresponds to an iteration. The values of sequences B and C in each row are snapshots of their values after step 3 of phase 3 in each iteration. The *Updates* column shows the order of updates of values in V that need to be performed so that $\mathcal{M}_{S'}$ reaches a stable state, in which each value is consistent with its associated definition.

In place of a step-by-step description of the phase, it is sufficient to highlight its significant features. Although the reference g is associated with a redefinition in the block K, it is not appropriate to update its value before the update of a and e are completed. Nodes such as g have non-zero Knuth counters are replaced by a marker (β') in step 2 of the phase.

The execution of phase 3 then iterates between steps 3 and 9. Sequence C contains all the direct dependents for the current focus of the iteration, vertex $head(B)$. For any iteration n with focus node $a = head(B)$, each element x from sequence C that has a Knuth counter of $kc(a)+1$ is appended to the end of sequence B. This is the case only if x, which is dependent on a through being a member of sequence C, can be directly updated after the update of a. There should be no dependents of other vertices to take into consideration prior to this update. At the end of each iteration, the head of sequence B is removed.

The termination condition for the phase and the algorithm is that the sequence B is empty. This occurs when the final node to be updated in the phase has an empty sequence C.

4.4 Issues of Complexity with DM Models

Given the abstract DM Model for dependency maintenance and the algorithm for its update, it is possible to define a measure of complexity by counting the number of updates of values[6] required for a particular dependency structure. Section 4.4.1 describes some simple characteristics of the measure r_S with reference to illustra-

[6]Updates count recalculations and the setting of values that have no associated dependencies.

tive examples. Two more elaborate case-studies are presented in which well-known sequential algorithms are represented in a dependency structure. The first of these maintains the maximum and minimum of a set of values (Section 4.4.2); the second maintains a sorted sequence of numbers (Section 4.4.3). In these studies, the number of updates required to bring a DM Model up-to-date are tabulated.

4.4.1 A Complexity Measure for Dependency Structure

To be able to analyse the efficiency of the execution of scripts from their DM Model representation, there needs to be a method for counting the optimal number of updates that need to be performed for one state transition. This is analogous to counting the number of a particular type of operations (multiplication, compare/exchange etc.) are carried out in a conventional algorithm. This section focuses on how dependency structures influence the maximum number of updates required to carry out a single redefinition in a DM Model state.

Given a DM Model $\mathcal{M}_S$ with $n = \beta - \alpha$ vertices in its associated dependency structure, the measure r_S (introduced in Section 4.3.2) is an upper bound on the number of updates needed to update V via a single redefinition[7].

Four scripts Q, R, S and T, each with associated dependency structures with four vertices ($n = 4$), are shown in Figure 4.7. This figure illustrates the variety of dependency structures that can exist for a small value of n. For instance, the dependency structure may not be a tree nor be connected.

For each of the dependency structures in the figure, a solid dot represents a vertex in the structure that, if redefined, would require the maximum possible number of updates of values in one state transition. Note how the value of r for a structure depends on both the height and width of the structure.

Figure 4.8 sheds further light on the characteristics of the measure r_S. The

[7] A block of redefinitions K with size $\parallel K \parallel = 1$.

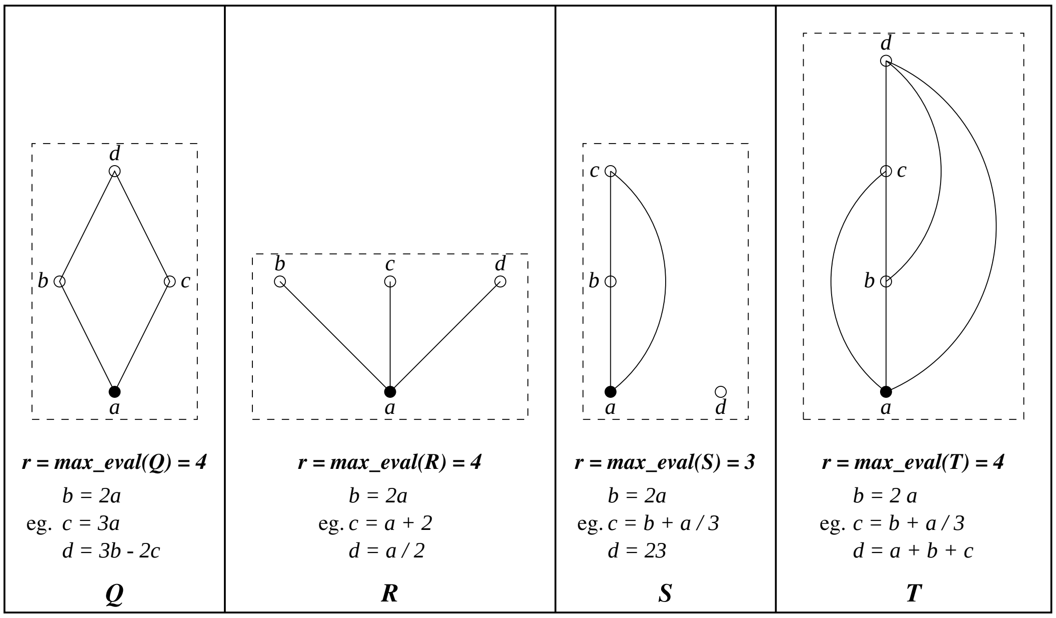


Figure 4.7: Possible patterns of dependency for four vertices.

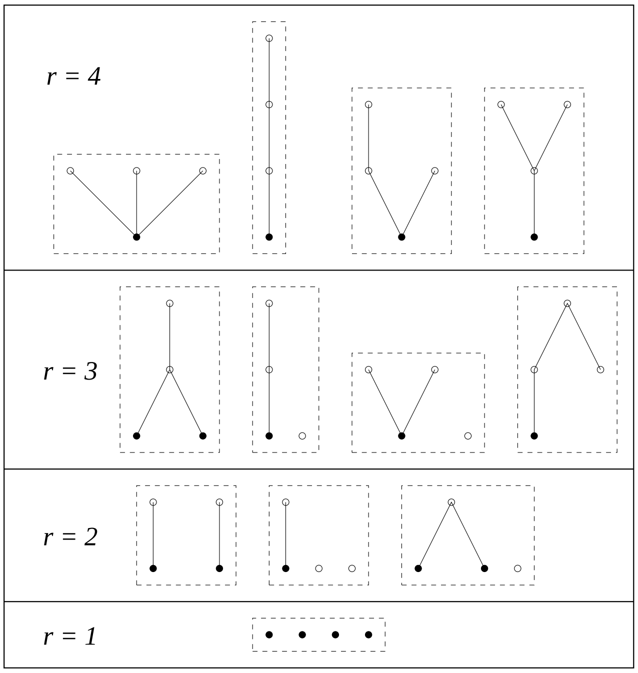


Figure 4.8: All possible patterns of dependencies for scripts with four definitions.

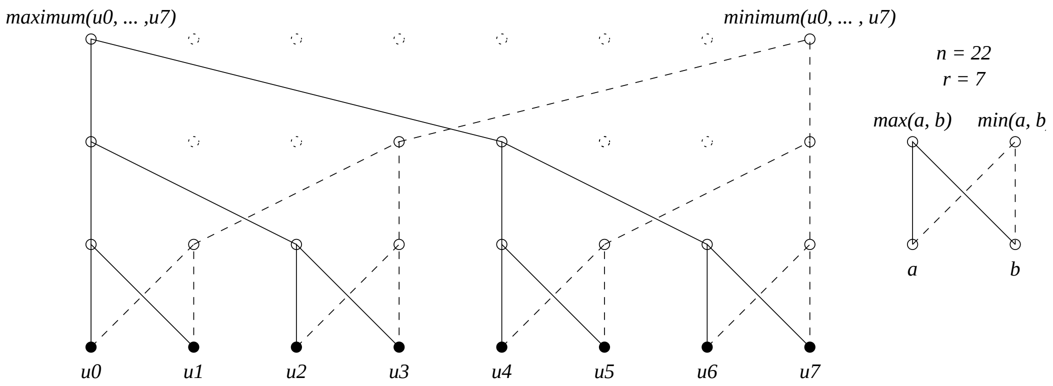


Figure 4.9: Dependency structure for finding minimum and maximum.

figure shows all the possible topologies for dependency structures associated with a script with four definitions, in which there is at most one path between each pair of vertices. As in Figure 4.7, vertices whose redefinition invokes the maximum number of updates are shown as filled circles. Future research will be directed at developing a theoretical framework for studying the measure r and associated dependency structures.

4.4.2 Case-study: Structure for Minimum and Maximum

This section examines a dependency structure developed to represent the dependency between a set of data and its minimum and maximum elements. It is appropriate to use the framework of the DM Model for this purpose, since the task of finding the maximum and minimum element of a set in a high-level definitive script with compound data types for values can be transformed to the same task with integers in a DM Model representable script. (As an application of such a structure consider determining the maximum dimensions of an enclosing bounding box for a group of geometric shapes as specified in a definitive script.)

Figure 4.9 shows a dependency structure that is associated with a script of definitions S that finds the maximum and minimum value of a set of integer values $\{u_0, \ldots, u_7\}$. The DM Model of script S has a maximum number of updates for one

Number of redefinitions	Minimum updates	Maximum updates	Minimum comp./exch.	Maximum comp./exch.
1	7	7	3	3
2	8	12	3	5
3	13	15	5	6
4	14	18	5	7
5	17	19	6	7
6	18	20	6	7
7	21	21	7	7
8	22	22	7	7

Table 4.7: Bounds for number of updates and compare/exchange operations for redefinitions - finding minimum and maximum value of a set.

redefinition of $r_S = 7$ in a structure with $n = 22$ vertices. In general, for a set of values $\{u_0, \ldots, u_m\}$ with script S_m for finding the minimum and maximum values (m is a power of 2), the maximum number of updates for one redefinition is r_{S_m}, where

$$r_{S_m} = 2\log_2 m + 1 \tag{4.23}$$

The number of vertices n_{S_m} in the dependency structure for S_m in this general case is

$$n_{S_m} = 3m - 2 \tag{4.24}$$

A conventional algorithm for finding minimum and maximum values requires a minimum of $m - 1$ compare/exchange operations. To update the values of a dependency structure when all m values are modified takes $2m - 2$ updates. These updates can be paired into $n - 1$ *min* and *max* operations that are similar to the conventional compare/exchange operations. If the minimum and maximum values are initially up-to-date, then if only one value is altered through redefinition of an element of the set $\{u_0, \ldots, u_{m-1}\}$, the block redefinition algorithm would update r_{S_m} values. This is the equivalent of $\log_2 m$ compare/exchange operations.

Table 4.7 shows the bounds for the number of updates for finding minimum and maximum. Each row corresponds to the size of the block of redefinitions for elements in the set $\{u_0, \ldots, u_7\}$. This table is constructed for the dependency structure depicted in Figure 4.9. The bounds for the minimum and maximum number of compare/exchange operations that the updates correspond to are also shown. The table shows that, when the block of redefinitions is smaller than the entire set, the block redefinition algorithm requires fewer updates than are required by the conventional algorithm for finding the minimum and maximum values for the whole set.

4.4.3 Case-study: Structure for Sorting

A similar analysis to finding the minimum and maximum of a set of values can be applied to sorting. A sorted sequence of values can be observed to be dependent on an unsorted set of values. Figure 4.10 shows a dependency structure that represents a script of definitions S for the Batcher bitonic merge sort [GS93] for a sequence of eight values $(u_0, \ldots, u_7)$. The sequence of sorted values is $(s_0, \ldots, s_7)$, where s_0 is the minimum and s_7 is the maximum value from the original sequence. If there is one redefinition of a value in the sequence then the upper bound for the number of updates required is $r_S = 35$.

In general, the number of compare/exchange operations required to sort a sequence of length m (a power of 2) using a Batcher bitonic sorting network is $m \log_2 m$. The total number of updates required to sort a sequence in a dependency structure is $2m \log_2 m + m$. This is the same as the number of vertices in the dependency structure.

Table 4.8 shows the maximum and minimum bounds for the number of updates and compare/exchange operations for the sorting structure illustrated in Figure 4.10, using the block redefinition algorithm. The updates are tabulated by row with reference to the size of the block of redefinitions $\| K \|$.

For small values of $\| K \|$, fewer updates and compare/exchange operations

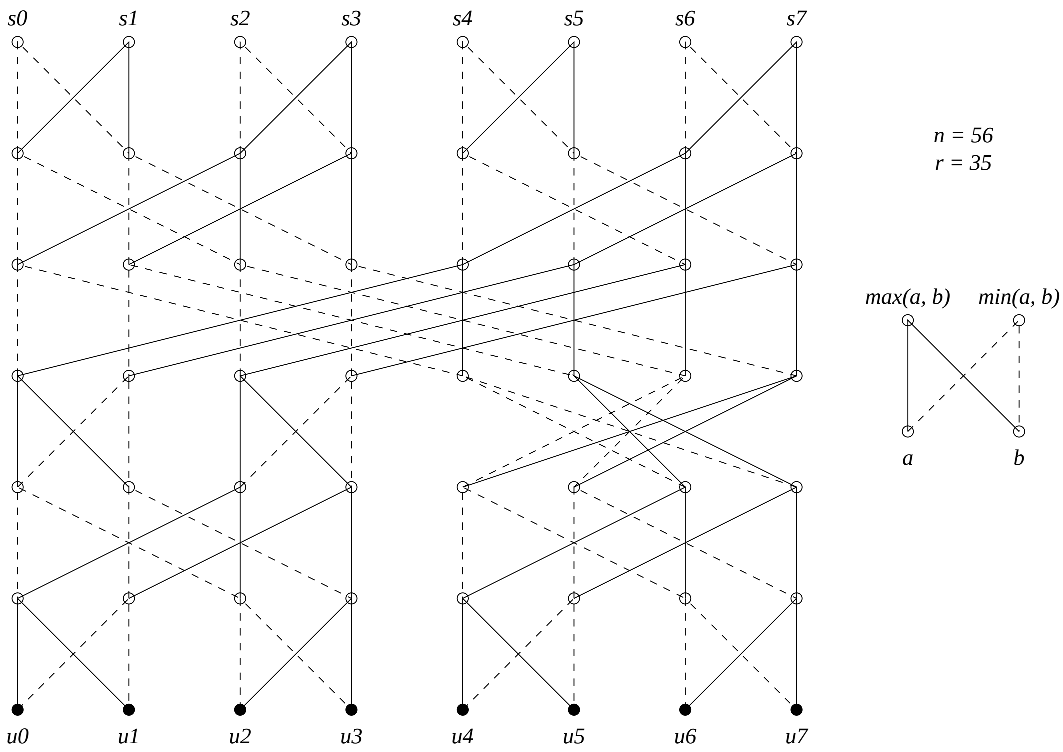


Figure 4.10: Dependency structure for sorting.

Number of redefinitions	Minimum updates	Maximum updates	Minimum comp./exch.	Maximum comp./exch.
1	35	35	15	15
2	36	46	15	22
3	39	49	16	23
4	40	52	16	24
5	51	53	23	24
6	52	54	23	24
7	55	55	24	24
8	56	56	24	24

Table 4.8: Bounds for number of updates and compare/exchange operations for redefinitions - sorting a sequence of values.

are required to sort the sequence of values. In the internal state of the DM Model, the intermediate values between the unsorted and sorted sequence are stored. Provided that the compare/exchange operations are computationally expensive relative to the ordering activity performed by the block redefinition algorithm, the definitive programming approach will be more efficient than a conventional procedural sort. Modulo the ordering activity, the block redefinition algorithm for dependency structure is always as least as efficient as the complete procedural sort.

The two case-studies demonstrate how dependency structure can be used as a form of algorithm. In the same way that the order of procedural execution of an algorithm can be optimised, the choice of dependency structure can affect the efficiency of state transition. The Batcher bitonic merge sort is chosen for the example as it is close to optimal for sorting. If bubble sort is used for the construction of a sorting dependency structure, then the number of updates required to complete the sort increases. Initial investigations into the representation of algorithms by dependency structure suggest that algorithms that are well-adapted for parallelisation are best suited for conversion to scripts of definitions. There are also clear connections between maintenance of dependency structures and dynamic algorithms.

Chapter 5

The DAM Machine

5.1 Introduction

The DM Model presented in Chapter 4 is a model for dependency maintenance for representing dependencies between one data type $\mathcal{Z}$. This chapter considers the question of whether it is possible to implement the DM Model, with its potentially infinite length value representation, on a computer system using multiple finite length integers. Motivated by the fact that computer systems use memory store that is made up words containing binary bits, with state either on or off, to represent all data of all types at the lowest level of representation, it seems appropriate that a n-bit word can be considered analogous to the data type $\mathcal{Z}$ in an implementation of the DM Model. This chapter presents the design for a dependency maintenance tool written in assembly code that maintains indivisible relationships between words in an area of computer RAM store. The tool is the called the *Definitive Assembly Maintainer Machine* (DAM Machine) and the area of words in store maintained consistent with its definition is called the *definitive store*.

The first section of this chapter (Section 5.2) explains the way in which the DM Model is implemented by the DAM Machine. The relationship between the mathematical model and the implementation is explained by relating the layout of store in the DAM implementation to the mappings and sets of the DM Model. The implementation uses the block redefinition algorithm[1] to keep the values in store consistent with their definitions. The DAM Machine is accessed through programming language function calls allowing a programmer to use an area of definitive store in their applications. The process for creating and accessing the store is explained.

The implementation design requires the programmer of a definition based model to decide how to represent observables as words in definitive store memory. It is necessary to consider the location in memory of a word and how the pattern of bits is significant to a higher-level data type, in the manner required for standard low-level assembly coding. The dependency maintenance in a script model is taken as close to the processor as possible in DAM Machine implementation, creating an efficient dependency maintaining virtual machine, without the need of an additional runtime interpreter or translator. Issues of data representation in the DAM Machine are discussed in Section 5.3 along with the stages of conversion of an existing definitive script notation into a tool that uses the DAM Machine to maintain its dependency.

A translator of the DoNaLD notation [ABH86], the notation for line drawing integrated into the *Tkeden* package, that uses DAM as its dependency maintainer instead of EDEN is presented in Section 5.4. This was developed by Allderidge as part of his third year undergraduate project work. This implementation uses the same parser as the DoNaLD notation and generates an intermediate script of definitions in a format that is suitable for execution on the DAM Machine. The translation process is explained with reference to the stages of the construction of the DM Model. Efficiency claims for the DAM Machine concept are supported by a comparison of timings between the *Tkeden* tool and a DAM Machine representation

[1]See Section 4.3.3 of Chapter 4.

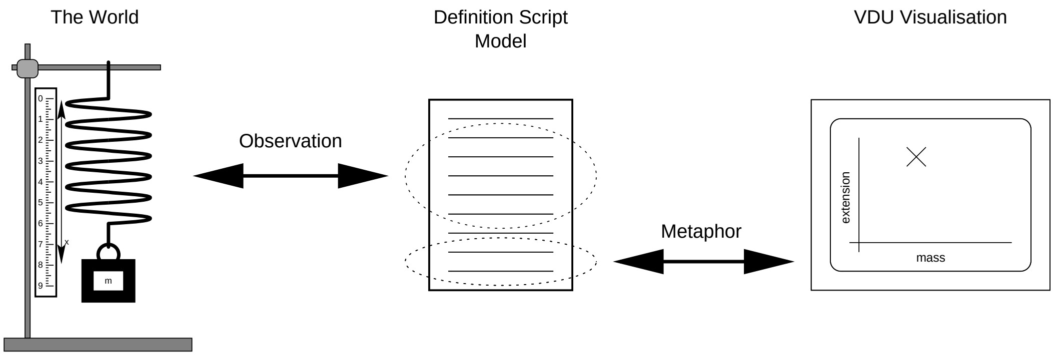


Figure 5.1: Script describing a model and its visual metaphor.

of the same DoNaLD model.

The *Vehicle Cruise Control Simulator* [BBY92], developed to run in *Tkeden*, demonstrates how at any point of interaction with the interpreter of an empirical model, it is possible to either refine the script of definitions of the model or the metaphor for its visualisation. For example, it is possible to redefine the current speed and observe the change to the speedometer or to redefine the geometry of the speedometer at any point through the interaction. Figure 5.1 shows how this idea could be extended to a low-level maintainer such as DAM. The current output of a visual display of a computer system is held in the memory of the computer itself. The location of a set of pixels forming a cross pattern on the screen can represent the observed mass / extension relationship of a spring in an observed model that is represented as a script of definitions. By placing the pixels of the display in an area of memory where words are maintained consistent to their definition, the *definitive store*, it is possible to build a direct link between the model as observed and visualised. An illustration of this process for geometry is presented in the last section of the chapter (Section 5.5).

5.1.1 Scics Machine

An experimental precedent for the DAM machine is the em Scics Machine, prototyped by Simon Yung [Yun96]. Scics combines the UNIX-based spreadsheet "*sc*" with a computational machine simulator. This simulator was developed at the University of Warwick with the aim of assisting with the task of teaching of computer science. Machine code and the machines memory resides in cells of the spreadsheet. Dependency maintenance is carried out by the spreadsheet and this takes precedence over machine execution.

Yung used the *Scics Machine* to implement heapsort [AHU82] in such a way that, for example, spreadsheet dependencies serve to update the heap conditions when values on the heap are exchanged[2].

5.2 From the DM Model to DAM Implementation

A DM Model represents a script of definitions transformed from a high-level as a low-level script with one data type, the integers $\mathcal{Z}$. The DAM Machine is an implementation based on the DM Model where there is also only one data type, a word of computer *Random Access Memory* (RAM) store. A state i of a DM model $\mathcal{M}_S = (A, F, D, V)$ can be related to the state of the DAM Machine memory. Different computer architectures use different lengths of words, each containing n-bits. On an n-bit architecture there are 2^n possible values for each word. The computer system on which the DAM Machine is implemented is the Acorn Risc PC that uses an ARM RISC processor [Fur96] and is a 32-bit architecture. For this architecture, each word has a total of 2^{32} possible values.

The set of references A of the DM model is replaced by a range of addresses in store for the DAM Machine. These locations in memory are known as *definitive*

[2]Beynon recently developed a *Tkeden* interactive model for the heapsort algorithm and this is described in [Bey98b].

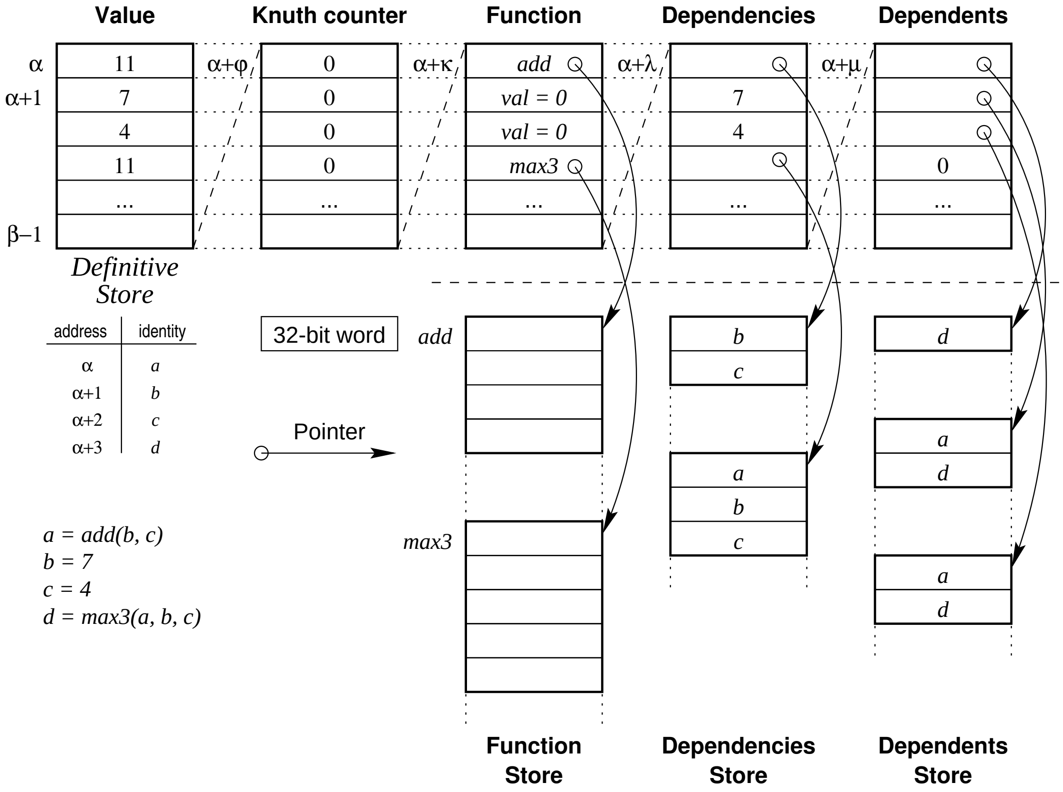


Figure 5.2: Design of the memory layout for the DAM Machine.

store, a continuous section of memory with a range bounded by low address α and high address $\beta - 1$. Unlike the DM Model, the implementation design requires that this set is the same size throughout the life of a script and that all new definitions are actually redefinitions of existing definitions. The initial state of the DAM Machine is that all references are associated with the DAM equivalent of the $value_0$ function, have no dependents and an initial value of 0. Figure 5.2 shows the design of the memory layout for the DAM Machine, with a definitive store starting at address α and ending at address $\beta - 1$ in the block of words in the top left hand corner labelled *Value*.

An example script shown at the bottom left hand corner of Figure 5.2. Above this script is a small table showing how the identifiers in the script are mapped to addresses in the DAM Machine. The rest of the figure is a graphical representation of the memory layout for the implementation of the DAM Machine based on the example script.

In a DAM Machine with a definitive store of size $s = \beta - \alpha$, additional store is required to represent the dependency between addresses. Four other areas of store exist that are the same size as the definitive store. Each area is used to hold information used to maintain dependencies between the words in definitive store. For each word, these are used to represent:

- pointers to the code for the operator used to maintain an indivisible relationship between the value of the word and the value of other words in the definitive store (*function* in the figure);
- pointers to a list with elements that are locations in store on which the word depends (*dependencies* in the figure);
- pointers to a list with elements that are locations in store that depend on the value of this word (*dependents* in the figure);
- a *Knuth counter* for use in the block redefinition algorithm. See Section 4.3.3 for details of the *Knuth counter* value in the block redefinition algorithm.

For a store of size s, there must be at least five times this number ($5s$) of words of memory to represent the dependencies in the store. An area of heap memory is required to store the lists of pointers (the *dependencies store* and *dependents store* in Figure5.2) and an area of store for the assembly code used to maintain dependencies (the *function store* in the figure).

The block of store for the Knuth Counters in the DAM Machine model are offset from the addresses of definitive store by φ ($\varphi \geq s$), starting at low address $\alpha + \varphi$ and ending at high address $\alpha + \varphi + s - 1$. The current Knuth Counter for a value stored at address $\alpha + j$ is definitive store is itself stored at address $\alpha + \varphi + j$.

The other three blocks of memory are organised in a similar way, with the function block offset by κ, the dependencies block offset by λ and the dependents block offset by μ[3].

The block for function pointers corresponds to the mapping F in the DM Model, which maps any reference to the operator that is used to calculate the value connected with the reference. All mappings in the DM Model are members of the set $\mathcal{F} = \{f \mid f : \mathcal{Z}^* \to \mathcal{Z}\}$, comprising functions from a sequence of values of data type $\mathcal{Z}$ to a value of type $\mathcal{Z}$. The DAM Machine equivalent is a block of code that calculates the return value of one word from a sequence of words. These mappings can be mathematically represented as members of the set $\{f \mid f : [0, 2^{32} - 1]^* \to [0, 2^{32} - 1]\}$. More details of DAM operators can be found in Section 5.2.1.

The *dependencies* for each word in definitive store, references to the values that the word depends on that are also the arguments to the associated defining operator, are stored in a block offset by λ from α. For a value in definitive store at address $\alpha + j$, a pointer to a singly linked list for the dependencies for that value is stored at address $\alpha + \lambda + j$[4]. This list, stored in the operating systems heap, contains addresses for the values in definitive store that are dependencies in the same way that the mapping D associates a reference with a sequence of references that are dependencies in the DM Model. If the pointer in the operator block is equal to 0 for a particular associated value in definitive store, the value has no dependencies and it is assumed that its operator is equivalent to the DM Model $value_X$ operator. In this case, X is stored in the associated dependencies block instead of a pointer to a list.

The *dependents* block of store at an offset of μ from α has no analogous

[3]The relationships between φ, κ, λ, μ and s are as follows: $\kappa \geq \varphi + s$, $\lambda \geq \kappa + s$, $\mu \geq \lambda + s$.

[4]In Figure 5.2, the *dependencies store* and *dependents store* are represented as blocks of memory. In implementation, these are singly linked lists, the order of which determines the order of the sequence of arguments to the operator.

component in the DM Model. For a word in definitive store at address $\alpha + j$, a pointer to a singly linked list of the addresses for dependents of that word, those that have the word as a direct dependency, is stored at address $\alpha + \mu + j$. This list is maintained to assist the efficiency of the algorithms that update the store with redefinitions by representing a doubly linked dependency structure. If the value at an address in the block is 0, the associated word in definitive store is a *root node* at the top of a dependency structure for the definitions represented.

5.2.1 DAM Machine Operators for Definitions

A definition in a script prepared for the DM Model is of the form

$$identifier = operator(arguments).$$

In the DAM Machine, an operator is analogous to a block of assembly code that is passed address references to words that represent the *arguments* sequence of a definition, in the current states of processor registers. The code should calculate a value based on looking up the values stored at these addresses and place this calculated value back into a register `R0`. For any definition, the DAM Machine ensures that before control is passed to the block of code for an *operator*, the correct references are stored in the registers to represent the *arguments* and that the value remaining in the register is stored into the address in definitive store for the *identifier*.

For example, consider the definition "*a = add(b, c)*" shown in Figure 5.2. The operator *add* should be a block of assembly code instructions that lookups the values stored at the addresses for b and c (that are $\alpha + 1$ and $\alpha + 2$ respectively), load these into registers, add the values in the two registers together and move the result into register zero. The block of code is passed a pointer of where to return control after execution and should end with an appropriate branch instruction to do this. The ARM code for an *add* operator is shown below[5].

```
.add LDR R2,[R0]   ; Load R2 with defn. store value at R0 (b)
     LDR R3,[R1]   ; Load R3 with defn. store value at R1 (c)
     ADD R0,R2,R3 ; Set R0 equal to R2 + R3 (b + c)
     MOV PC, R14  ; Return control to the DAM Machine
```

A programmer who is implementing a DAM Machine dependency driven part of an application must write the assembly code for these operators. The code should be consistent with their chosen data representation over the 32-bit words supported by the DAM Machine. More details on the data representation strategy are presented in Section 5.3.

5.2.2 Redefinitions and the Update Mechanism

A programmer who wishes to use a DAM Machine in an application is required to allocate sufficient memory to store the number of definitions they wish to represent in definitive store by choosing appropriate values for α, β, φ, κ, λ and μ. The initial state of the DAM Machine definitive store is that every value is set to 0, every associated Knuth counter is set to 0, every function pointer and dependencies pointer is 0 to represent the special $value_0$ operator and every dependent is 0. They must also place into memory the assembly code for the operators, as described in Section 5.2.1. To redefine a value in store at an address that lies between α and $\beta-1$, the programmer must load into the processor's addresses the parameters shown in the list below and then branch to the "`addtoq`" subroutine of the DAM Machine.

`R0` The address of the value in definitive store to be redefined.

`R1` A pointer to the start of the block of operator code used to calculate the value implicitly defined by the definition for the associated identifier in `R0`. If `0` is in

[5] In the current implementation, the references to addresses that are dependencies are passed to the operator subroutine in registers `R0` up to `R10` and the return pointer for the subroutine in `R14`. This enforces an implementation specific limit on the length of the argument list to a maximum of 11.

this register then the special $value_X$ operator is assumed, in other words the value at the address in `R0` is explicitly defined to be X in register `R2`.

`R2` If `R1` is loaded with a `0` then `R2` is the value of X for the special $value_X$ operator. Otherwise, register `R2` should be loaded with a pointer to the value in definitive store that is the first argument in the definition argument sequence for the redefinition.

`R3` to `R12` The pointers stored in these registers represent the elements with indices 2 up to 11 of the argument sequence for the redefinition. These pointers should be addresses in definitive store that are dependencies for the value stored at the address in `R0`. The first value of `0` in order through the registers signifies the end of the argument sequence and the order of pointers in increasing register index is the same as the order of the sequence for the redefinition.

This causes the DAM Machine to place the redefinition of the queue of redefinitions awaiting that next branch to the DAM Machine "`update`" subroutine. This queue is equivalent to the set K in the DM Model. Figure 5.3 shows two redefinitions for the script used in the example in Figure 5.2, "*d = add(a, c)*" and "*c = 3*". Directly below these redefinitions in the figure are the registers that should be set before the call to the "`addtoq`" subroutine of the DAM Machine. The effect if this subroutine is to store these register values in an area of memory internal to the DAM Machine.

When the "`update`" subroutine is called for a DAM Machine in state i equivalent to a DM Model $\mathcal{M}_S = (A, F, D, V)$, this queue of definitions is considered the set K of redefinitions. The block redefinition algorithm is used to update the state of the DAM Machine to state $i + 1$. If any the redefinitions on the queue introduce cyclic dependency into the model, an error is reported and the previous state i of the DAM Machine and definitive store is restored. Along the bottom section of Fig-

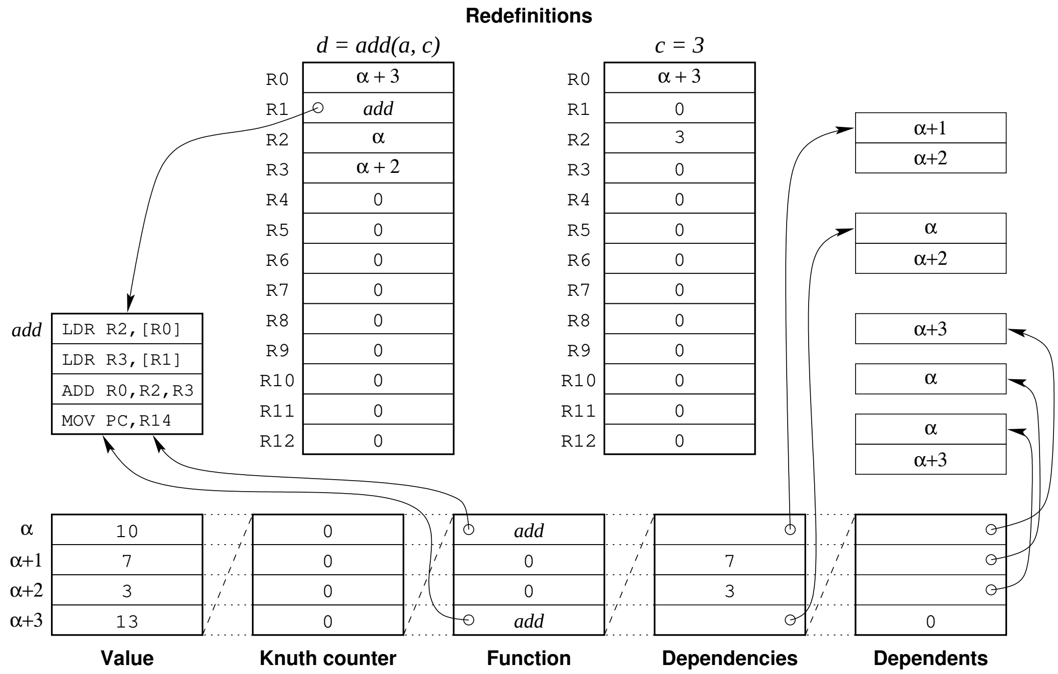


Figure 5.3: Two redefinitions and their effect on DAM Machine Store.

ure 5.3, the successful call to the "`update`" method due to the two definitions results in the definitive store and other associated blocks of store. The lists of dependencies and dependents are shown down the right hand side of the figure and the block of code for the operator is shown on the left hand side.

It is up to the programmer to ensure that they choose a suitable layout for definitive store for their application. If a programmer wishes to make reference to a value in store, they simply need to load it into a register (using the "`LDR`" instruction). No DAM Machine mechanism stops them storing a value into this area of memory again (using the "`STR`" instruction). The definitive store only remains definitive if the programmer of an application that includes such a store only changes the store through calls to the "`addtoq`" and "`update`" subroutines of the DAM Machine.

5.3 Data Representation

As already demonstrated in this chapter, one word in computer store can represent either a data value, a pointer to an address in store or a processor instruction. For one data value alone there may be many different interpretations of the pattern of bits, for example: a positive integer, a positive or negative integer with the high bit equivalent to a minus sign, a 32-bit floating point number representation, two positive 16-bit integers, four characters in a string, thirty-two separate Boolean values. The representation of data types in one word is discussed in Section 5.3.1. It may be that 32-bits does not provide a programmer with sufficient level of detail for values that their application requires. For example, values that represent pictures, strings of characters, 64-bit *double* arithmetic and so on. Multi-word data representations are discussed in Section 5.3.2.

5.3.1 One Word Data Representations

The DAM Machine does not *know* through its internal data structures the data types of the values in definitive store. It provides no internal representation for the types of stored data like a lookup table and performs no type checking on whether a sequence of arguments is of appropriate type to pass to an operator subroutine. A programmer has to ensure that all redefinitions are appropriate to their application and type check arguments for their operators prior to calling the "`addtoq`" subroutine of the DAM Machine. This can be considered as a weakness of the DAM Machine as it does not support these features common to a high-level language. However, the intention is that using the DAM Machine is at the same low-level as a programmer developing assembly language code is used to dealing with. The DAM Machine provides an architecture for dependency maintenance that the developer of a high-level language notation can exploit as the object of a compiler

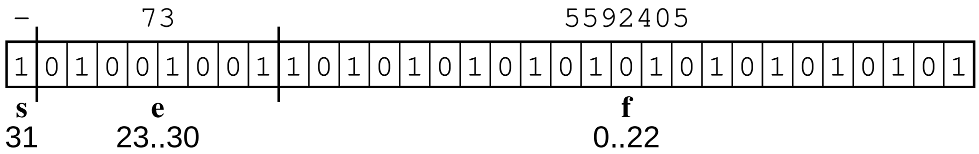


Figure 5.4: IEEE single format representation for a floating point number.

tool.

As long as a programmer implements their operators appropriately and consistently, it is possible maintain dependencies between their chosen data representations if 32-bit words are satisfactory for their application. Figure 5.4 shows a technique for the representation of floating point numbers in 32-bit words. The representation is consistent with the IEEE standard for the representation of single length floating point values [IEE85]. The word of 32-bits is split into 3 bit fields: the bit fraction **f** between $0 \ldots 22$ with 0 the least significant, the biased exponent **e** between $23 \ldots 30$ and the high sign bit **s**. For the range $0 < \mathbf{e} < 255$, the floating point value represented by the bit pattern is $-1.0^{\mathbf{s}} \times 2^{\mathbf{e}-127} \times (1 + \frac{\mathbf{f}}{2^{23}})$. The number represented in the example in the figure is $-1.0^{1} \times 2^{-54} \times (1 + \frac{5592405}{8388608}) \simeq -9.25185 \times 10^{-17}$.

For an application where every word in a definitive store is represented by a single length floating point value and all the operators map between sequences of floating point numbers and a floating point number, this data representation is perfectly adequate. If an application has more than one data representation, such as 32-bit integers combined with floating point, then a programmer needs to keep their own lookup table to determine whether the value at a particular address in store is floating point or integer. They will need subroutines for arithmetic operations on floats and different subroutines for the same operations on integers, for example "*add_float*" for floating point and "*add_int*" for integers.

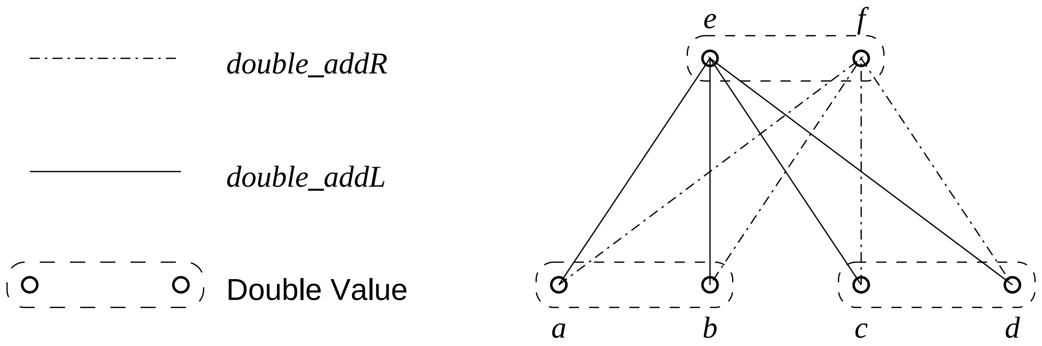


Figure 5.5: Operators for double length floating point operations.

5.3.2 Multi-word Data Representations

What happens when one 32-bit word is not sufficient to store accurately an item of data in an application? Multiple words in definitive store can be joined together to create longer ones in multiples of 32-bits at the discretion of a programmer. The operators represented by blocks of assembly code in the DAM Machine map to a value through the registers of the processor[6]. The registers have a limited length of 32-bits so it is not possible for one operator to maintain dependencies for more than one machine word in definitive store. For a two word (64-bit) data representation, the programmer has the option to implement two specialised operators, one that maps to the highest bits $(32 \ldots 63)$ and one that maps to the lowest bits $(0 \ldots 31)$.

Consider the example shown in Figure 5.5 that represents the addition of two double numbers and maps to another double. Three pairs of nodes in a dependency structure are shown, each pair representing a double length floating point value. The double represented by the pair (e, f) is defined to be the addition of the doubles represented by pairs (a, b) and (c, d). To do this, two operators rather than one are required, one for bits $32 \ldots 63$ of the result called "*double_addL*" and one for bits $0 \ldots 31$ called "*double_addR*". A script of definitions for this dependency structure is shown below.

[6]See Section 5.2.1.

e = *double_addL(a, b, c, d)*
f = *double_addR(a, b, c, d)*

The management of the location of words in definitive store is the task of the programmer of an application. It may be necessary to keep track of both the types of each word and also which words combine to form the value of a definition. For compound data structures such as arrays it becomes necessary to allocate a block in store in which to represent the array data. As operator code is passed a reference to a location in memory, a lookup operator can be defined to map from a base address of an array in definitive store p and an offset q for the q^{th} element of the array, to the value stored at the combined address $p+q$. The code for an operator to do this is shown below[7]:

```
.lookup LDR R3,[R1]     ; Load array offset value into R3 (q)
        MUL R3,R3,#4    ; Calculate word address offset (4q)
        LDR R2,[R0]     ; Load array base value into R2 (p)
        LDR R0,[R2,R3]  ; Load R0 with value stored at p + q
        MOV PC,R14      ; Return control to the DAM Machine
```

In this code, the value stored in definitive store at address p is a pointer to the beginning of an array that is also in definitive store. The dependency structure can maintain dependencies between pointers as a data types as well as between data values. If arrays that change their size are required in an application, the programmer should implement code that is capable of shifting blocks of definitions around, either through reading the definitive store and all its associated data and repeatedly branching to the "`addtoq`" subroutine, or by moving the definitions around themselves ensuring they remain consistent to the internal representations of the DAM Machine.

The representation of data in DAM Machine words requires a programmer to think carefully about the pattern of bits they are to use inside the words to

[7]Note that the addressing of words in RAM is 8-bit "byte by byte". The boundaries of words occur every four bytes, hence the multiply by four line in the code.

hold a data value of a certain type. It does not provide a user friendly interface or parser for a language as the *Tkeden* tool does. Instead, it provides a *Virtual Machine* similar to the Java Virtual Machine [MD97] that can be the target of a compiler's output. With appropriate management of definitive store, the translation process from a high-level language to DAM Machine representation can be used to implement dynamically extendable, open-ended tools for modelling based on scripts of definitions. Unlike the *Java Virtual Machine* and *Tkeden*, the DAM Machine uses only assembled code and replaces the procedural interpretation of *bytecodes*[8] by the maintenance of a block of store consistent with its definition. The move away from interpreted code during the maintenance of dependency leads to more efficient execution of definitive script models.

5.4 The DoNaLD to DAM Translator

The DoNaLD to DAM translator, including a parser for the DoNaLD notation that uses the DAM Machine as its back end dependency maintenance mechanism, was developed by Allderidge as his final year undergraduate degree project [All97], with further detail available in [ABCY98]. The tool developed takes the parser for DoNaLD notations as integrated into the *Tkeden* interpreter and replaces the mechanism for the maintenance of dependency inside the interpreter with the DAM Machine rather than definitions translated into EDEN. The efficiency saving of using the dedicated DAM Machine can achieve up to 20 times better performance in the animation of DoNaLD scripts through redefinition than *Tkeden* running on a similar speed processor.

The process of definition translation from the high-level DoNaLD notation to DAM through an intermediate notation is described in Section 5.4.1. This process

[8]This is done by a huge `case` statement on the Java Virtual Machine and *Tkeden*.

is similar to the stages of the construction of the DM Model from the observations of scripts are presented in Section 4.2.1 of Chapter 4. The degree to which open-ended action is possible with the tool is discussed. Two case-studies for DoNaLD are presented in Section 5.4.2, with a comparison of their performance in animation between the existing *Tkeden* tool and the DAM Machine.

5.4.1 The Translation Process

The DoNaLD to DAM translator is a parser for the DoNaLD notation that operates in two phases. The first phase translates a DoNaLD file into *pseudo-DAM code*, in which each line represents a definition for the DAM Machine. This code is then translated in the second phase by another simple parser into the DAM Machine internal representation by placing appropriate values in the processor registers and then branching to the "`addtoq`" subroutine of the DAM Machine, followed by the "`update`" subroutine. The pseudo code contains some references to special operators that not only represent a dependency but also take an action within the body of their code to alter the graphical image associated with the DoNaLD script, such as a line or a circle.

Figure 5.6 shows an original segment of DoNaLD script for one line in the top window ("`simple`"), its representation is *pseudo-DAM code* in the bottom left hand window ("`scriptfile`") and its graphical representation in the small bottom right hand window ("`simple [default]`"). The figure demonstrates phase one of the translation process. Most lines of the pseudo code contain an operator name for a block of code for an operator in store, followed by an identifier for the definition and a sequence of identifiers for arguments to that operator. Two special operators "`seti`" and "`setf`" are equivalent to the DM Model $value_X$ function. They treat the second argument as the explicit value X for the identifier in the first argument. The other operators shown in the figure are described in the following list. The

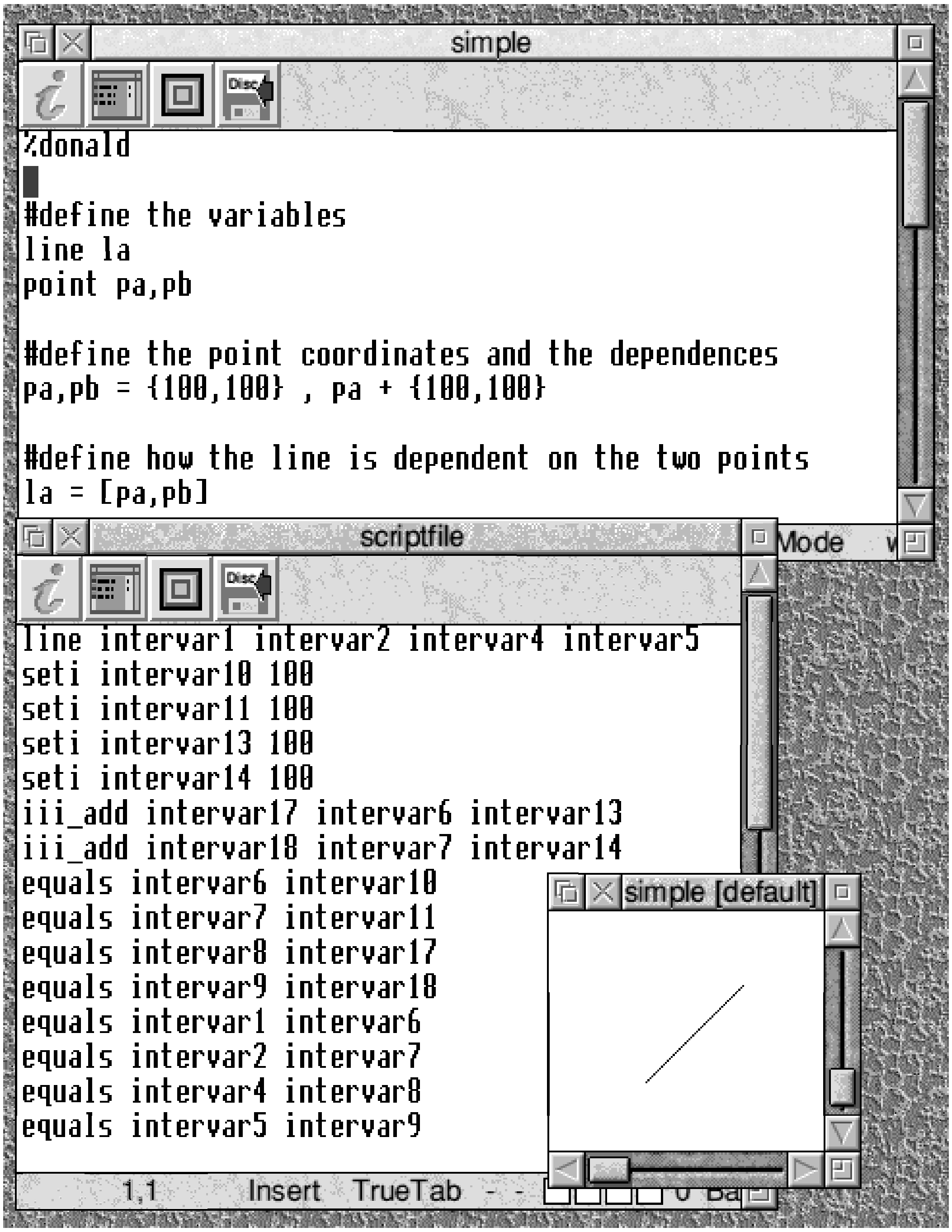


Figure 5.6: Phase 1 of translating a DoNaLD definition for a line into a DAM Machine representation.

strings that start "`intervar`" and end with a number n are identifiers for integer values automatically generated by the parser.

`line` An operator that maps from four integer values to a line defined by two end points. The line is not given an identifier name in the pseudo code and their is no internal one word representation for a *line*. Code inside the block for the operator `line` causes a line to be drawn on the screen.

`equals` A simple operator that defines the value of the identifier second along the line to be equal to the value associated with the identifier third along the line.

`iii_add` This operator adds two integer values together, the values at the locations for the third and fourth identifiers on the line, and places the result in the memory location associated with the second identifier on the line. The "`iii`" stands for an operator that maps from two integers to an integer. Equivalent operators, such as "`fif_add`" for adding an integer and a floating point value and returning an integer, are available to the pseudo code and the DAM Machine implementation of DoNaLD.

The process of translation follows the stages of transformation of a high-level script to make it DM Model representable, with the additional concerns relating to splitting the representation of complex data types such as points into a component representations suitable for store in DAM Machine words. An outline of the stages is given in the enumerated list below, for each definition in a DoNaLD script (a line of the form "*identifier* `=` ...").

1. If the definition contains an expression on its right hand side, build an expression tree. If the expression contains arithmetic for data types that require more than one word to represent them in the DAM Machine, these should be

split into trees for the arithmetic on each component (e.g. split arithmetic for points into two expression trees, one for each dimension).

2. For each parent node of the tree, create an associated reference name (e.g. "`intervar1`"). Each parent node in the tree will become a definition in the pseudo-DAM code. This process removes function composition from definitions and identifies which of the DAM operators can be used to maintain dependencies.

3. If the definition in the DoNaLD is for an explicit value that can be represented by one word in definitive store, create an associated reference name for it. If it is represented by two words in definitive store, create two reference names for it and so on. Store the association between the high-level name and the internal reference name in a lookup table. For implicit definitions, associate the reference root(s) of the related expression tree(s) with the high-level identifier.

4. Write out the pseudo-DAM code, in which every reference name has an associated line in the code and has an appropriate operator for the DAM Machine. The pseudo code contains only internal reference names and no identifiers from the high-level script. Ordering in this script is not important. Explicit values, the leaves of expression trees or sub-components of explicit value definitions in the high-level notation, are defined using the "`seti`" and "`setf`" operators.

Figure 5.7 shows the result of this process for the DoNaLD script shown in Figure 5.6[9]. The dependency structure in the DoNaLD script is represented in the top left hand diagram and the dependency structure in the pseudo-DAM code, and hence the DAM Machine internal representation, is shown on the right hand side. The reference names in the pseudo-DAM that are associated with the identifiers in

[9] A node in the dependency structure in Figure 5.7 labelled *i1* corresponds to the `intervar1` identifier in Figure 5.6.

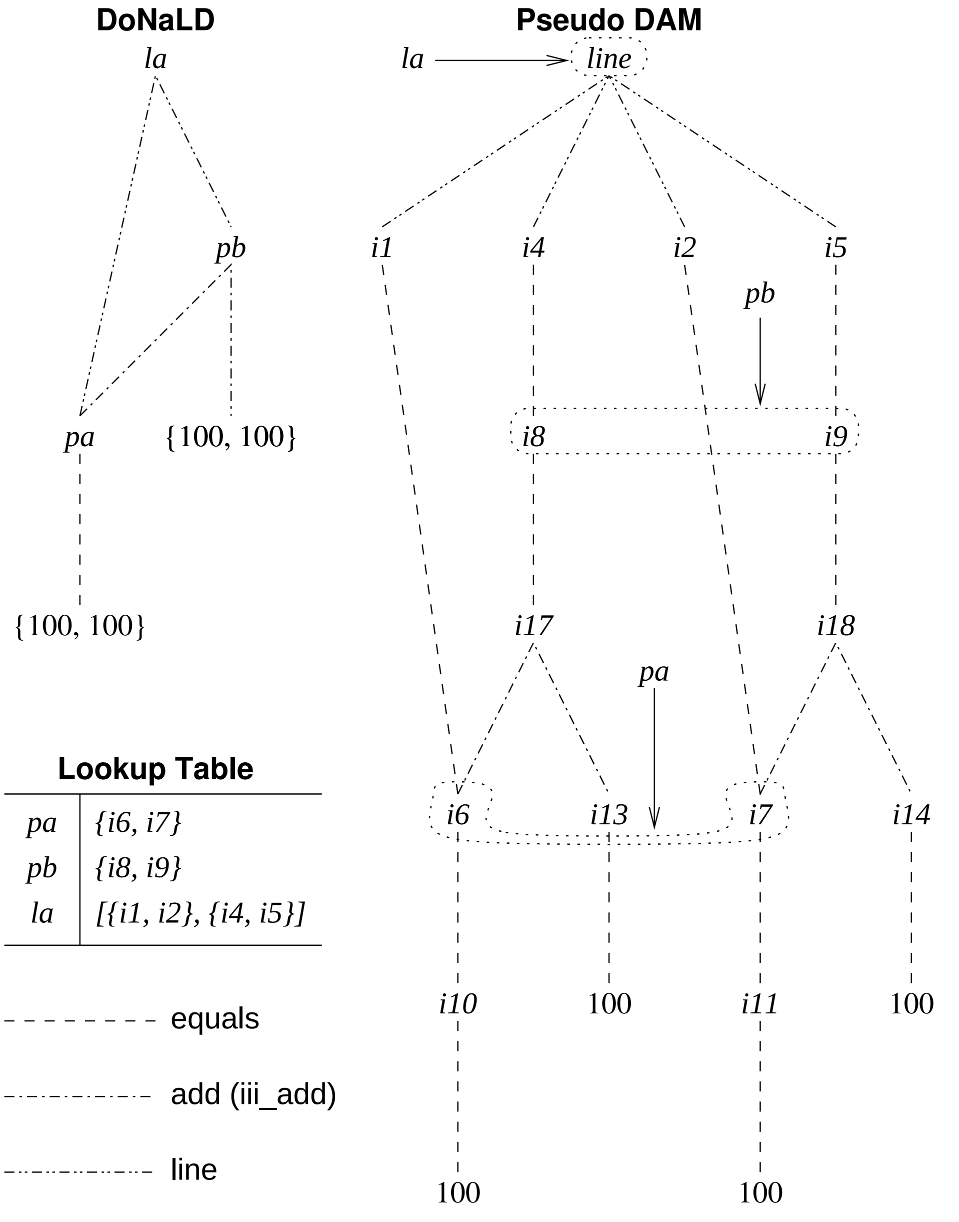


Figure 5.7: Dependency structures for the translation of DoNaLD to pseudo-DAM code.

the DoNaLD script are shown both in the lookup table in the figure and where they are circled by dotted lines in the dependency structure. The steps corresponding to the translation process above for this example script are shown in the following list, with pa and pb considered in order as two separate definitions "$pa = \{100,\ 100\}$" and "$pb = pa + \{100, 100\}$".

Firstly, the process of translation for pa. The expression trees[10] are generated by the existing DoNaLD parser.

1. Identifier pa is declared as a point that is represented by two words in store in the DAM Machine and is therefore split into two expression trees. These are "x = *100*" and "y = *100*".

2. Internal reference names are associated with the parent node of each tree. To do this, x is replaced by *i10* and y is replaced by *i11*.

3. Two reference names (*i6* and *i7*) are created in the translator's lookup table to represent the components of the point pa. The values at the roots of the trees are defined to be equal to the values associated with these new reference names, *i6* equals *i10* and *i7* equals *i11*.

4. The pseudo-DAM code is prepared and is of the form:

```
seti intervar10 100
seti intervar11 100
equals intervar6 intervar10
equals intervar7 intervar11
```

Secondly, the translation process for pb:

[10]Expression trees are represented in the text in the form "*root_node = child_node*" or "*root_node = l_child operator r_child*". The children are either explicit numerical values, existing reference names in the lookup table or sub-trees surrounded by square braces "*[]*".

1. Identifier *pb* is declared as a point that is represented by two words in store in the DAM Machine and is therefore split into two separate expression trees. As the high-level expression depends on the value of another definition *pa*, the expression trees reference the lookup table to find the reference names of the component values of *pa*. The two trees are: "*x1 = i6 + [x2 = 100]*" and "*y1 = i7 + [y2 = 100]*".

2. Internal reference names are associated with the parent node of each tree. To do this, *x1* and *x2* are substituted for by *i17* and *i13* respectively, similarly *y1* and *y2* by *i18* and *i14*.

3. Two reference names (*i8* and *i9*) are created in the lookup table to represent the components of the point *pb*. The values at the roots of the trees are defined to be equal to the values associated with these new reference names, *i8* equals *i17* and *i9* equals *i18*.

4. The pseudo-DAM code is prepared and is of the form:

```
seti intervar13 100
seti intervar14 100
iii_add intervar17 intervar6 intervar13
iii_add intervar18 intervar7 intervar14
equals intervar8 intervar17
equals intervar9 intervar18
```

The result of the production of the pseudo-DAM code is an appropriate set of definitions that represent the same dependencies in the DAM Machine internal store as in the DoNaLD script. In the current implementation, it is possible to re-define explicit integer and floating point values in this structure using the "`addtoq`" and "`update`" subroutines through a graphical user interface, but it is not possible to make other kinds of new definition or redefinition[11] because the phase 1 translator terminates after reading one file to allow the phase 2 translator to convert the pseudo-DAM code into store. This is technical problem that will need to be researched in the future.

5.4.2 DAM Machine Performance

The DAM Machine is implemented on one platform only and this is a different platform from that on which *Tkeden* is implemented. The performance tests carried out in this section are based on timing the animation of DoNaLD scripts by repeatedly increasing the value of a defining explicit parameter. The DoNaLD code for these scripts is presented in Appendix A.1. The results presented here are not intended as a direct comparison of performance between systems because of their physical differences, but they do demonstrate the potential for fast dependency maintenance on a stand alone computer system running the DAM Machine. When the same DoNaLD script is executed on hardware of similar speed, the DAM Machine can produce a smooth animation where *Tkeden* models appear jerky and slow.

Figure 5.8 shows the DoNaLD graphical output from the script of an engine[12]. The position of the pistons in the script is defined to depend on one variable "`lup`" that determines the current rotation of the engines crank shaft. When an appropriate point is reached in this rotation, the next firing of a spark plug in the firing-order sequence is represented by the temporary appearance of a circle at the top of the cylinder concerned. By repeatedly increasing the `lup` parameter, the location of the pistons inside the engine's cylinders can be animated.

Table 5.1 shows timings for the animation of the engine script and another

[11] Such as all implicit definitions, explicit definitions of points and lines etc..

[12] DoNaLD source for the engine is available is Section A.1 of Appendix A.1. The image is a screen snapshot of the DAM Machine version.

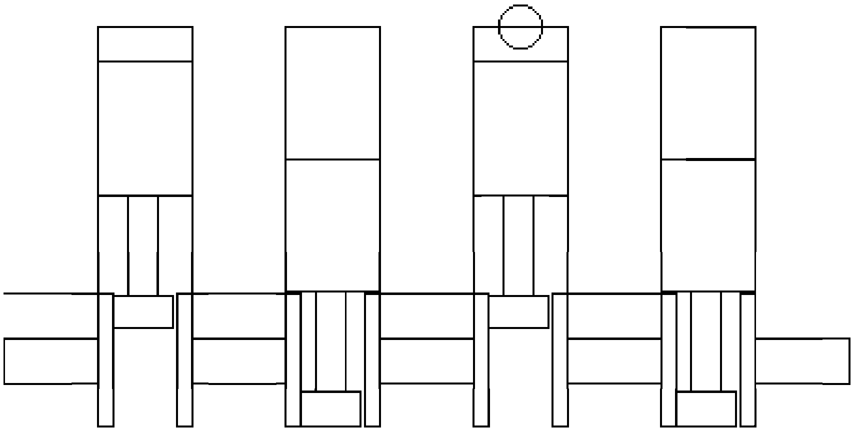


Figure 5.8: Graphical output from the DoNaLD *Engine* script.

Script	ARM710 CPU running RISC OS windows.	ARM710 CPU single tasking.	CY76601 CPU (SPARCstation 2) running X windows.
Engine	26	13	262
Shapes	21	15	172

Table 5.1: Comparative timings for the animation of DoNaLD scripts on an ARM710 processor and a SPARC CY76601 processor.

DoNaLD test script called *shapes*[13]. Timings were carried out for the DAM Machine on an Acorn *Risc PC 700* with an ARM710 processor, clocked at 40MHz and capable of 36 MIPS[14]. A similar specification SUN Microsystems SPARCstation 2 with a CY76601 processor clocked at 40Mhz and capable of 28.5 MIPS was used to run *Tkeden*. Both systems were tested in multi-tasking mode displaying graphical windows interfaces with no other significant tasks running simultaneously.

To show the potential for a single tasking application that uses dependency and is based on the DAM Machine, timings are included for the DAM Machine executing the DoNaLD script with no graphical windows interface running. The application writes directly to the screen memory through low-level system calls. The range and increment value for the parameters repeatedly updated were the same on the Acorn system and the SUN system. Timings shown are an average of three separate timing experiments.

The table shows how the multi-tasking versions of the animation of the engine script produced a factor of 10 improvement for the DAM Machine over *Tkeden* and a factor of 20 to the single tasking version. For the shapes script, which contains more trigonometry and less line drawing, the speed increase is not quite so impressive with a factor of 8 speed increase for the multi-tasking SUN over the Acorn, and a factor of 11.5 for the multi-tasking SUN and the single-tasking Acorn. The visual difference between the two systems is marked. The single-tasking Acorn produces a smoother animation than the slower *Tkeden* model.

[13]DoNaLD source available in Appendix A.1.

[14]The MIPS unit is a measure for the number of Million Instructions Per Second a processor is capable of executing.

5.5 Linking Observation with Visual Metaphor

The concept of linking the state of the internal model with the state of the display of the screen through dependency maintenance was first introduced in the introductory section of this chapter. Figure 5.1 shows how the current state of an experiment to demonstrate Hooke's Law can be represented by a cross positioned on a computer screen. In this section, a case-study is used to show how a block of pixels located in DAM Machine definitive store can be maintained consistent with their individual definitions. They share the same definitive store with other definitions for the model that they represent.

In this case-study, an area of 1600 machine words in definitive store is chosen to represent a 40 by 40 grid of pixels for display on the screen. The value stored at each machine word is 0 if the pixel is black and 1 if the pixel is white. Each pixel is implicitly defined to depend on the value of the function representation[15] of a two dimensional geometric shape, which is sampled at a particular x coordinate and y coordinate. If the pixel is outside the specified shape, as indicated by a negative value of the function representation at the point given by (x, y), it is defined to be black. Otherwise, the pixel is white to represent that it is inside or on the boundary of the shape.

The function representation of two dimensional geometric shapes is achieved in the DAM Machine by special operators that map defining parameters for a shape to a pointer to a subroutine block of machine code. This code evaluates the function representation at a point (x, y) dependent of these parameters. In the case-study, there are three such operators: *circle* for a planar circle shape parametrised by centre and radius, *reactangle* for a rectangle shape parametrised by minimum and maximum x and y coordinates, *cut* parametrised by two pointers to other func-

[15]The function representation for geometric shape is introduced in Section 3.4.2.

tion representations for shape where one shape is cut away from the other (the set theoretical difference operator). The value of (x, y) is passed to the subroutines in registers `R0` and `R1` respectively.

The function representations used in the case-study and implemented as blocks of ARM assembly code for this case-study are specified below.

circle Definition of the form "$circle(r, c1, c2)$". A circle shape centred at point $(c1, c2)$ with radius r. The function representation for the shape f for any point (x, y) is given by the formula

$$f(x,y) = r^2 - (x - c1)^2 - (y - c2)^2 \tag{5.1}$$

rectangle Definition of the form "$rectangle(xmin, xmax, ymin, ymax)$". The rectangle has sides parallel to the x and y axis of the coordinate space, with minimum coordinate along the x-axis of $xmin$ and maximum coordinate $xmax$ etc.. The function representation for the shape f for any point (x, y) is given by the formula

$$f(x,y) = ((x - xmin) \cap (xmax - x)) \cap ((y - ymin) \cap (ymax - y)) \tag{5.2}$$

In this formula, the binary infix operator "$a \cap b$" is used to represent set intersection and in the implementation this is achieved in function representation of shape by finding the minimum value of the two arguments a and b.

cut Definition of the form "$cut(f1, f2)$". The shape represented by a cut definition is the material of the shape with function representation $f1$ with the material of the shape with function representation $f2$ removed. The function representation f for the shape of the cut material at any point (x, y) is given in the formula

$$f(x,y) = f1(x,y) \cap -f2(x,y) \tag{5.3}$$

Each pixel on the screen is defined by the "*evaluate*" operator, which maps a pointer to a function representation *f1* and a sample point (x, y) to 0 or 1. The *evaluate* operator is given by the equation

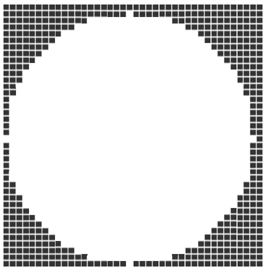
$$evaluate(f1, x, y) = \begin{cases} 0 & \text{if } f1(x, y) < 0 \\ 1 & \text{if } f1(x, y) \geq 0 \end{cases} \tag{5.4}$$

In the case-study shown, the index of a pixel in the array of pixels is used as the point (x, y). The definition of the pixels is set up by code similar[16] to the double *for* loop shown below:

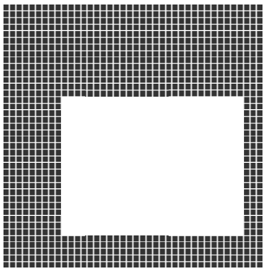
```
for i = 0 to 39 do
  for j = 0 to 39
    do
       "p_ij = evaluate(f1, i, j)"
    done
  done
```

The case-study illustration in Figure 5.9 contains one circular shape, one rectangular shape and has one shape made by the cutting operator. If the function representation for a circle with centre $(19, 19)$ and radius 19 is defined as *f1* with the evaluate operator in the code above, the pixels represent the image labelled "*Circle*" in Figure 5.9. The image is an expanded screen snapshot image of the pixels as displayed by the DAM Machine model. The "*Rectangle*" displayed in the top right hand side of the figure: it has the parametrisation $xmin = 9$, $xmax = 36$, $ymin = 5$ and $ymax = 25$. The rest of the figure concerns a "*Cut*" shape. A section of definitive script that could be used to describe the dependency between the cut shape and the circle and the square is shown below.

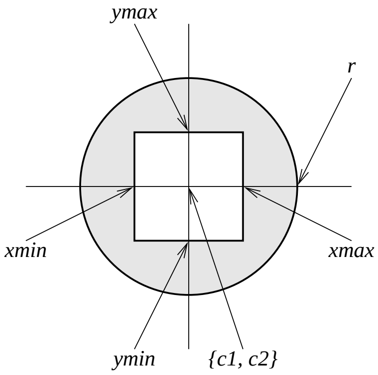
[16]The code used to set up these definitions for the case-study is written in the *BBC Basic* language. The actual code does not illustrate the point of the example as well as the pseudo code presented.



Circle



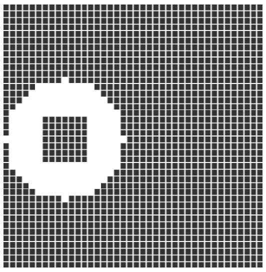
Rectangle



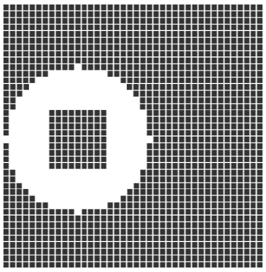
Cut diagram

$$
\begin{aligned}
r &= 9 \\
c1 &= r \\
c2 &= 19 \\
halfr &= half(r) \\
xmin &= subtract(c1, halfr) \\
xmax &= add(c1, halfr) \\
ymin &= subtract(c2, halfr) \\
ymax &= add(c2, halfr)
\end{aligned}
$$

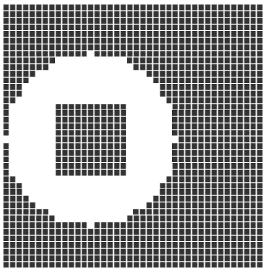
Script for *Cut*



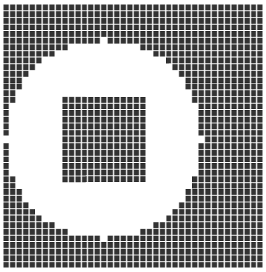
Cut at $r = 9$



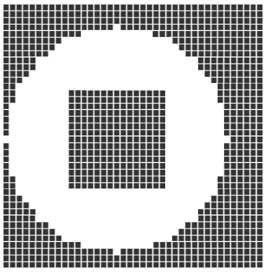
Cut at $r = 11$



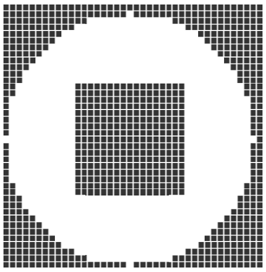
Cut at $r = 13$



Cut at $r = 15$



Cut at $r = 17$



Cut at $r = 19$

Figure 5.9: Direct dependency between screen and script models.

$$\begin{aligned} circ &= circle(r, c1, c2) \\ sq &= rectangle(xmin, xmax, ymin, ymax) \\ ct &= cut(circ, sq) \end{aligned}$$

This environment allows a user to experiment with models of rectangular shapes cut away from circular shapes. The definitions and diagram shown in Figure 5.9 illustrate some further relationships that can be introduced to the DAM Machine model. The geometry represented by the model is a circle that has a square, with sides the same length as its radius and centred at the centre of the circle, cut away from inside it. The sequence of six screen images shown at the bottom of the figure show the effect of redefining the value of r by calling the "`addtoq`" and "`update`" methods of the DAM Machine. In each case, the image changes through the update propagation through definitive store, not through a procedural sampling of the function representation of the cut. The time for this propagation through 1600 DAM Machine definitions on the 40MHz single tasking ARM710 processor running the DAM Machine is 0.52 seconds for each image.

This case-study demonstrates the concept of creating indivisible relationships between the screen and a model. This creates an open-ended environment in which a modeller can simultaneously construct the model, improve the model, create a visual metaphor for interacting with the model and fine tune the visualisation to their requirements. Visual operations, such as zooming and panning the image, and changes to the parameters to the model are achieved through the same redefinition mechanism. The state of machine words in definitive store remain consistent with their definitions through the DAM Machine and hence the state of the image also remains consistent with its definition. The better the quality of the visual metaphor for the model, the closer the experience of interaction with a model is to actual interaction with the model's referent.

Chapter 6

The JaM Machine API

6.1 Introduction

In some previous research work on definitive notations, prior to the implementation of the *Tkeden* interpreter, the practical emphasis was on the design and implementation of new definitive notations. These notations were specific to particular applications. The DoNaLD notation [ABH86] for line drawing and the SCOUT notation [Dep92] for screen layout were both developed with this aim in mind, as was Stidwill's partial implementation of the CADNO notation [Sti89]. For each notation, a translation process translates definitions in the application specific notation into EDEN definitions and data types through one-way communication along the UNIX command pipeline. In practice, interaction with scripts of definitions was inefficient because of the process of converting application specific data types into EDEN data types. This inefficiency was compounded by the slow process for the interpreted execution of EDEN code, in which translated data types typically require a myriad of EDEN interpreted operators and procedures to manipulate them.

The *Tkeden* interpreter does not support the command pipeline and, as a consequence, does not support the development of new application specific definitive notations[1]. This problem is the motivation for the *Java Maintainer Machine Application Programming Interface* (JaM Machine API), a general purpose programming toolkit. The purpose of this toolkit is to allow a programmer to develop and implement new definitive script notations with data types and structures appropriate to the application in question. Applications developed with the API use compiled code for actions and update of definitions. This leads to more efficient execution of notations than the translation and interpreted execution mechanism of EDEN.

The most significant features of the JaM Machine API, which contribute new possibilities for the implementation of tools that support empirical modelling, are:

- Independence of data structures from the operators used to define dependency relationships between them. This is enabled by the object-oriented representation of the data types and operators. The JaM Machine API is a general purpose toolkit for the construction and integration of application specific definitive notations.

- Support for multi-agent interaction with scripts of definitions. Each definition has associated read/write permissions, supported by usernames and passwords for the secure identification of agents.

- Use of the Java programming language, with the following benefits:

  - multi-platform execution of empirical modelling tools without the need for compilation;

  - access to the class libraries of the Java APIs that support multi-threading, networking, graphical user-interfaces and integration of applications into world-wide-web browsers;

[1] Although the previous versions of EDEN are still in existence, they are not maintained.

– dynamic extension of compiled code for data types and operators of an empirical modelling notation on-the-fly.

The JaM Machine API update mechanism is based on the DM Model presented in Chapter 4. The general purpose nature of the JaM Machine API is realised through the object-oriented analysis of definitive scripts. This analysis is presented in Section 6.2 of this chapter along with a description of the classes that make up the API and guidelines for using them. Support for multi-user interaction within a single script is built into the API, based on a novel approach to agency similar to that used to protect file access on a multi-user computer system. This approach is presented in Section 6.3.

Scripts in definitive notations created using the JaM Machine API are known as *JaM scripts*, these scripts are written in *JaM notations*. A JaM notation is intended to be seen as a kind of intermediate, lower-level code for a higher-level definitive notation. To implement a notation such as DoNaLD [ABH86] with JaM Machine classes would require a parser for DoNaLD that performs some transformation from the DoNaLD script to a JaM script. As the JaM classes form an API, they can be integrated with any other Java software to provide dependency maintenance components within that software. Mechanisms for the incorporation of the dynamic type systems and dependency maintenance within other software are described in Section 6.4.

The final section of the chapter (6.5) shows a simple example of a JaM notation for arithmetic that is integrated into an *Arithmetic Chat* server/client application. The JaM Machine API is not developed specifically to support geometry and it is not intended that this programming toolkit is in any way similar to a programming API for geometry, such as *Djinn* [BCJ+95, BCJ+97]. The JaM Machine API is a general purpose programming toolkit for dependency maintenance.

A major case-study that uses the JaM Machine API is presented in Chapter 7, in which the API enables the creation of a definitive notation for the modelling of three-dimensional *empirical worlds*. These worlds have data types and operators to describe three-dimensional solid geometry.

6.2 Object-Oriented Analysis of Definitive Scripts

In the same way that a definitive script was analysed in Chapter 4 for the explanation of the DM Model, the underlying concepts of the JaM Machine API are based on the analysis of definitive scripts. These observations lead to the description of classes of objects that encapsulate the underlying components and mechanisms of dependency maintenance. The JaM Machine API is a set of classes in a Java package called "`JaM`". Rather than the one data type $\mathcal{Z}$ over which dependency maintenance is considered in the DM Model in Chapter 4, data types in JaM notations are represented as classes of objects that extend `DefnType`, an *abstract class*[2] in the JaM package.

Figure 6.1 shows the relationships between classes in the JaM package. The classes shown are the public classes [CH96] and those that are required for a thorough explanation of the API. If a class is connected to and on the right of another class then it extends the class to the left. Classes shown in the diagram in a box surrounded by a dashed line are abstract classes.

The breakdown of definitive scripts into classes of the JaM package is described below. Where appropriate, reference is made to the JaM specific definitive script shown in Figure 6.2 to explain how the analysis of a script leads to such a classification of classes. JaM scripts have similar properties to DM Model repre-

[2] An abstract class is one which cannot be instantiated to become an object as it contains stubs for the body code of some of its methods. Classes that extend an abstract class must implement these stubs. See [CH96] for more details.

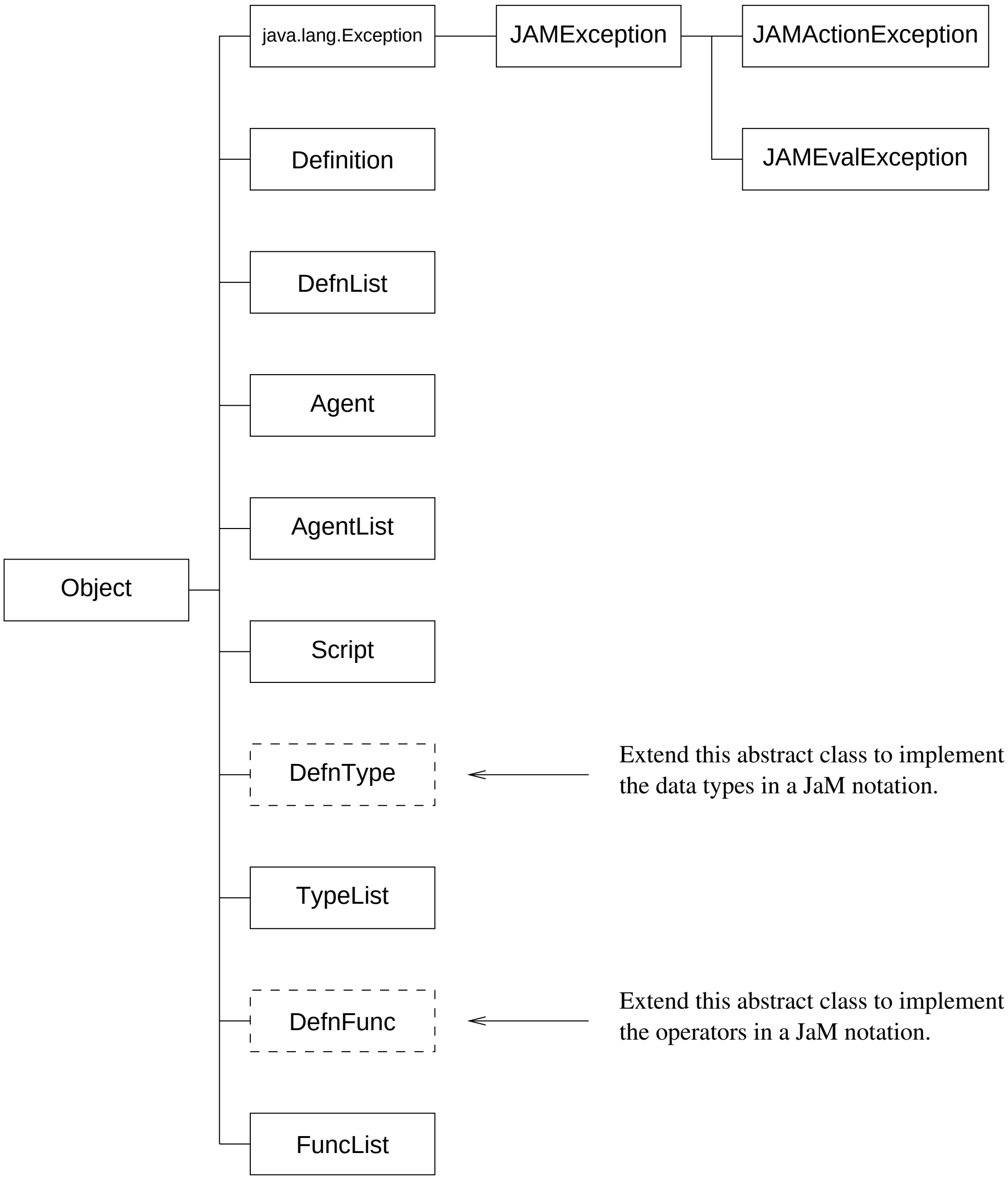


Figure 6.1: Class diagram for the JaM Machine API's package `JaM`.

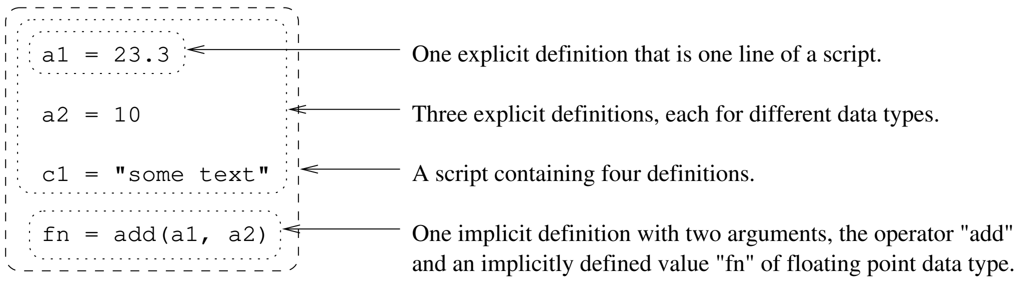


Figure 6.2: An example of a four definition JaM script in a definitive notation with its component features annotated.

sentable scripts (see page 104): operator composition is not allowed and arguments to operators must be other identifiers that already exist in the script. A programmer can implement a translating process to convert definitions from a notation with composition to a JaM notation using a parsing tool or a method similar to the *DoNaLD to DAM Translator* presented in Chapter 5 (Section 5.4). In contrast to this translation process, identifiers are not replaced by reference numbers as they are in Stage 3 of the DM Model transformation process (see page 107). Each definition in a JaM script is identified by a string.

`Script` A `JaM.Script` class represents a set of objects that describe definitive scripts, including their current state and the mechanisms to update that state. A script object for the example shown in Figure 6.2 would contain a list of four definitions, data types for integer, floating point and string values and one operator "`add`". Instances also contain a queue of redefinitions awaiting a call to their `update()` method, which has the effect of updating the state of the script consistent with the definitions on the queue. This class is described further in Section 6.2.4.

`Definition` An instance of the `JaM.Definition` class represents one line of a definitive script. In the script shown in Figure 6.2, each definition is either of the explicit form "*identifier* `=` *value*" or the implicit form "*identifier* `=` *operator* `(`*ar-*

guments)". For every definition object, there is a variable for the string representing the *identifier* (`a1`, `a2`, `c2` or `fn` in the figure), along with variables to store the current *value* associated wit the definition (`23.3`, `10` etc.) in the current state of the script and additional information such as who owns the definition. Additional information is associated with an implicit definition: a reference to the defining *operator* (`add` in the figure) and references to the definitions that make up the sequence of *arguments* (`a1` and `a2`) to the operator. Definitions are discussed further in Section 6.2.1.

`DefnList` A `JaM.DefnList` represents an indexed list of zero or more definitions. An instance of this class contains a doubly-linked list data structure for references to objects which are instances of the `Definition` class. These lists are used to represent the script and queue in a `Script` class, dependency structures in the `Definition` class and the arguments to an implicit definition.

The construct "`(a1, a2)`" in Figure 6.2 is a textual representation of a list of definitions. More information on how this class is accessed is presented in Section 6.2.3 which describes operators in JaM scripts.

`Agent` Every definition in a JaM script can be associated with an instance of a `JaM.Agent` class who is the agent owner responsible for the definition. The use of agency in the JaM Machine is presented in Section 6.3 with the `Agent` class itself described in Section 6.3.2.

`AgentList` Instances of `AgentList` contain doubly-linked lists of references to instances of the `Agent` class. `AgentList` instances are used to establish an access control mechanism with username and password databases, as described in Section 6.3.

`DefnType` The data types of JaM notations are created by the implementor of a notation by extending the abstract class `JaM.DefnType`. This process is described in Section 6.2.2 of this chapter. In the example script in Figure 6.2 the data types are floating point numbers, integer numbers and strings of characters.

`TypeList` Instances of `TypeList` contain doubly-linked lists of references to instances of classes that extend `DefnType`. This class is used to represent the current list of available types in an instance of the `JaM.Script` class. Three data types are illustrated in the example script. These three types would have a representation in the `TypeList`.

`DefnFunc` The operators of the underlying algebra over the data types of a JaM notation are created by the implementor of a notation by extending the abstract class `JaM.DefnFunc`. This process is described in Section 6.2.3. An operator in a JaM notation can be defined to map from the values of any sequence of arguments of the data types of that notation to the value associated with an identifier, that can also be of any type. This flexibility can be restricted to provide a more appropriate underlying algebra for the given type structure of an application specific notation. In the example script in Figure 6.2, the only operator is `add`, the argument list is a list of identifiers of ordinal types and the implicitly defined left-hand-side value for the *identifier* `fn` is a floating point value equal to `33.3`.

`FuncList` Instances of `FuncList` contain doubly-linked lists of references to instances of classes that extend `DefnType`. This class is used to represent the current list of operators available in an instance of a `JaM.Script` class. Only one data type is available in the example script and this would be the only element of the operator list in the script instance representing the example.

`JAMException` If an exception is caused by the execution of one of the methods in one of the other classes, a `JAMException` is thrown, requiring the code that called the method that there was a problem in its execution. More information on Java exception handling can be found in [CH96]. The exceptions thrown by methods of classes in the JaM package are tabulated in Appendix A.2.

The rest of this section of the chapter concentrates on the major components of the object-oriented analysis of definitive scripts. Where appropriate, the relationship between the DM Model presented in Chapter 4 and the JaM package classes is emphasised to highlight the use of the block redefinition algorithm (see Section 4.3.3) to update the state of JaM scripts.

6.2.1 Definitions

Two kinds of definitions are possible in a JaM notation, an implicit or an explicit definition. An explicit definition is of the form "*identifier* = *value*". When an explicit definition appears as a redefinition in a JaM script, its string description has the *identifier* token replaced by a name for the definition and the *value* replaced by a string of characters exactly describing the value of a data type supported in the JaM notation. From this string description, it is possible to infer which data type the `value` represents and to establish the data type associated with the *identifier*. Explicit definitions are like definitions in the DM Model that use the "$value_X()$" constant operator, which maps the associated value of any identifier to the constant value X.

An implicit definition in a script is of the form

identifier = *operator* (*arguments*)".

The *identifier* token should be replaced by a string name for the definition and the *operator* by a string name for an operator available in the current JaM notation. The arguments to the operator are a comma-separated sequence of other identifiers whose names exist prior to this implicit definition in the JaM script. Implicit definitions establish the value associated with the *identifier* by evaluating the *operator* with the values in the sequence of *arguments*. The data type associated with the *identifier* in the script given by an implicit definition depends on the ordering of the data types in the list of *arguments*.

Implicit definitions in a JaM script are similar to implicit definitions in the DM Model. For each identifier there is an associated operator in the same way that for a reference in the DM model there is an associated mapping (F) belonging to the set $\{f \mid f : \mathcal{Z}^* \rightarrow \mathcal{Z}\}$ (see Sections 4.2.1 and 4.2.5). Also, for every implicitly defined identifier in a JaM script there is an associated sequence of arguments similar to the mapping (D) that takes a DM Model reference to a sequence of references. Every identifier in a JaM script has a current value in the same way that every DM Model reference can be mapped to a current value (V).

The data fields in a `JaM.Definition` class are shown in Table 6.1. In this table, the Java types for the fields are shown in the left-hand column, the identifier given to the fields in the class in the central column and a description of each field in the right-hand column.

6.2.2 Data Types

Conventional parsing methodologies for high-level languages use context-free grammars. A similar process is required for the parsing of values in explicit definitions in scripts, although the grammars used in a JaM script are simple. As every explicit definition in a JaM script can be observed to be of the form "*identifier* = *value*", it is easy to put in place a mechanism to parse the characters before the equals sign as a string identifier. More complicated is the process to match the string of characters on the right hand side of the equals sign to a regular expression describing a

Java Type	**Name**	**Description**
`String`	`name`	The *identifier* for an instance of the class in a JaM script.
`DefnType`	`value`	The *value* for an internal representation of a definition in its current state.
`DefnFunc`	`f`	The *operator* used to implicitly define the value of an instance of this definition. Set to `null` if the definition is explicit.
`boolean`	`fExists`	A boolean which is `true` if an instance of the class is implicitly defined and `false` if it is explicitly defined.
`DefnList`	`dependencies`	List of object references to definitions on which the value of this definition depends directly. This represents the sequence of *arguments* to the implicit definition. Set to `null` if the definition is explicitly given.
`DefnList`	`dependents`	List of object references to definitions that directly depend on this definition.
`int`	`knuthCounter`	The Knuth counter used in the block redefinition algorithm.
`Agent`	`owner`	The owning agent of this definition is a JaM script.
`boolean`	`ownerRead`	A `true` value if the `owner` can reference this definition in another implicit definition, `false` otherwise.
`boolean`	`ownerWrite`	A `true` value if the `owner` can redefine this definition instance, `false` otherwise.
`boolean`	`allRead`	A `true` value if agents other than the `owner` can reference this definition instance in their implicit definitions, `false` otherwise.
`boolean`	`allWrite`	A `true` value if agents other than the `owner` can redefine this definition instance, `false` otherwise.

Table 6.1: Data fields of a `JaM.Definition` class.

particular type and to parse the value into some internal representation.

A data type in a JaM notation is described by a simple context-free grammar that matches a pattern of characters. Each regular expression establishes a class of values that can be characterised and uniquely recognised by their string representation. Java supports a number of primitive data types and compound object data types that may be a more suitable and space efficient representation for a value given by a string. For example, the most efficient way to represent a string of characters that matches with a regular expression for a string in a Java application is through the use of a string type object. If a value matches with the regular expression for an integer, then the most efficient way to store this value is as a variable of the `int` Java data type. For a particular data type classification, it may be possible to find a translation methodology from a string of characters to an internal efficient data representation and from the internal representation back to a string of characters.

Data types in a JaM notation are defined by a programmer by implementing classes that extend the abstract class `JaM.DefnType`. Extended classes should contain data fields to hold an internal representation of the data of the type available to a user of a JaM script. The class must implement a method to match or reject a string description of a value given in an explicit definition in a script (`recognise()`), to parse this string description into its internal data representation (`parseIt()`) and to convert the internal data representation back into a string consistent with the current state of the value (`printIt()`). The programmer of sub-class of `DefnType` should ensure that a string for a value produced by the conversion of the internal representation of the data can subsequently be parsed back into the internal representation.

The methods that a programmer must implement within a class (X) that extends `DefnType` are shown in Table 6.2. Each row corresponds to the name of a method (**Name**). The types of arguments passed to each method are shown in the column labelled **Passed**, the methods return type in the column labelled **Returns** and a description of the methods purpose in the column labelled **Description**.

Figure 6.5, located close to the discussion of its contents on page 219, presents code for a class that extends `DefnFunc` that is for the representation of an integer number data type in a JaM notation.

6.2.3 Operators

Only prefix operators feature in JaM notations. No infix or postfix operators are available. They are used to define the indivisible relationship between data values in a JaM script. Operators have a string name that corresponds to the name that they use in a script. JaM operators are conceptually similar to operators in the DM model. Every operator in the DM Model is a member of the set $\{f \mid f : \mathcal{Z}^* \rightarrow \mathcal{Z}\}$ for the one data type $\mathcal{Z}$. Each JaM script operator can conceptually map from any sequence of values of any data type X to a value of any data type Y.

A programmer defines new data types in a JaM notation by extending the abstract class `DefnType`, as described in Section 6.2.1. The string value of an explicit definition is converted through a method call into some internal data representation and there are methods to convert this back to a string representation again. Two mechanisms are available for a programmer to manipulate data by an operator:

1. It is possible to define JaM operators that convert the internal value representation of a sequence of arguments into strings again, perform an operation by manipulating these strings to form a new string. This can then be reinterpreted as a value of a data type that itself extends `DefnType`, by using the `recognise()` and `parseIt()` methods. This mechanism allows for the description of some novel operators, but it is slow and should be used sparingly.

2. Use the internal data representations of classes (fields) representing data types

Method	**Passed**	**Returns**	**Description**

X	`String`	X	A constructor method for an instance of X. The constructor method should always adhere to the following generic template for its code:
			`X(String name) {`
			`  super(name);`
			`  ref = 1372; // substitute real ref.`
			`}`

`makeNew`	—	`void`	A method that is used to return a new instance of the data type represented by class X. This method should always adhere to the following generic template for its code:
			`DefnType makeNew() {`
			`  return new X(this.name);`
			`}`

`recognise`	`String`	`boolean`	A method that returns `true` if the passed `String` object is matched as being a pattern of characters that can be converted into the internal data representation for the data type, otherwise returns `false`.
`parseIt`	`String`	`void`	A method that converts the characters represented by the passed `String` object into the internal data representation for the instance of the class for which the method was called. The method `recognise()` is guaranteed to have returned a `true` value for the same `String` prior to the invocation of this method.

`printIt`	`int,`
	`int,`
	`int`
		`String`	A method that converts the internal data representation for the instance of the data type X for which it was called, into a `String` representation of the value. The three passed integers give hints to the method on how best to format this value string.

`action`	—	`void`	A method that can contain some procedural code that is executed every time a value of this data type is redefined or updated. The simplest acceptable implementation of the method is an empty block "{ }".

Table 6.2: Methods that must be implemented for a JaM data type class X that extends `DefnType`.

directly.

For every implicit redefinition, the data types in the sequence of arguments need to be checked to see that they are appropriate for the operator. This process also determines the type associated with the identifier on the left-hand-side of the definition. Any type change by the redefinition of an existing definition d have to he propagated through the dependency structure to type check all the operators that define the dependents of d ($dependents(d)$ in the DM Model).

Operators in JaM notations are implemented by extending the abstract class `DefnFunc`. The programmer must implement the methods shown in Table 6.3, which include an argument sequence checking mechanism (`typeCheck()`) and a method for the evaluation of the value mapped to by the operator (`f()`). Figure 6.6, located close to the main dicussion of its contents on page 221, shows some annotated example code for a class that extends `DefnFunc` and therefore represents an operator in a JaM notation.

Both methods `typeCheck()` and `f()` are passed references to a `DefnList` object that represents the arguments sequence for the operator in a JaM script. Two public mechanism are available to access the `DefnList` class. The first is a public integer variable `length` in the class that is the length of the arguments sequence. The second is a method `arg(`n`)` that returns the n-th element of the arguments sequence as an instance of `DefnType` that represents the current value for the that argument. As an example of this method, to find the reference number for the type of the second argument in sequence `dl`, a programmer can use the method call `dl.arg(2).ref`.

Method	**Passed**	**Returns**	**Description**

F	`String`	F	A constructor method for an instance of F. Typically a programmer will only instantiate one of these objects per JaM script. The constructor method should adhere to the following generic code template.
			
			`F(String name) {`
			`  super(name);`
			`  ref = 1372;  // Reference optional`
			`}`

`typeCheck`	`DefnList`	`DefnType`	This method is used to check that the passed `DefnList` is a sequence of data types appropriate to the operator. If it is, the method should return an empty instance of an object of the data type mapped to by the operator F for the given sequence of types, otherwise the method should return `null`.
`f`	`DefnList`	`DefnType`	This method carries out the evaluation of the mapping from the data values in the argument list `DefnList` to the implicitly defined value for the associated identifier. A programmer can rely on the fact that the method `f` is called only after the method `typeCheck()` has returned a non `null` object reference. The value returned should be a new instance of a class that extends `DefnType` with its internal data representation set to values consistent with the operator as used in the JaM script. Also, this returned object should be an instance of the same class returned by the `typeCheck` method.
`action`	`Definition`	`void`	A procedural action that is executed directly after the method `f()` is called to update the implicit value of an associated *identifier*. The action is passed a reference to the current state of the `JaM.Definition` instance for that *identifier*.

Table 6.3: Methods that must be implemented for a JaM operator class F that extends `DefnFunc`.

6.2.4 Scripts of Definitions

A script of definitions containing n lines can be represented by a `DefnList` class of length n. A JaM script is written in a particular JaM notation with data types that can be represented by classes that extend `DefnType` and operators by classes that extend `DefnFunc`. The states of a definitive script can be represented in a class that contains fields for: the script of definitions, a queue of pending redefinitions, the notation data types, and the notation operators over these data types. A mechanism for updating the current state of the script is also required.

A `JaM.Script` class contains the data fields shown in Table 6.4. This includes: the current JaM script of definitions (`theScript`), a queue for redefinitions (`queue`), a list of available types for the script (`tlist`), and a list of available operators over these types (`flist`) that make up the underlying algebra for JaM notation in which the JaM script is written. All these data fields are private to a script and can only be accessed through method calls.

Scripts change only through redefinition. If a programmer chooses to use a JaM script as a component of their application the first procedural step is that they must create a new script. Then they should decide what data types and operators the scripts that they are to use require and add these to the component type lists and operator lists of their script. The script initially has no definitions and no definitions waiting in the queue for update, similar to the initial state of the DM Model[3]. The next step in interacting with a new script is to add definitions to the definition queue. Once an initial queue is established, this list of definitions should be considered as the update for the initially empty main script so that the queue of definitions becomes the first state of this main script of definitions. This process is subject to checks for typographic errors in the definitions on the queue, for cyclic

[3]See Section 4.2.5.

Java Type	**Name**	**Description**
`DefnList`	`theScript`	The current state of the JaM script of definitions represented by an instance of the `Script` class.
`DefnList`	`queue`	A list of redefinitions that will be used for the next call to the `update()` method. This will update the current state of definitions in the list "`theScript`".
`TypeList`	`tlist`	A list of data types available in the current JaM notation for the JaM script represented by "`theScript`". These data types are represented by instances of classes that extend `DefnType`.
`FuncList`	`flist`	A list of operators available in the current JaM notation for the JaM script represented by "`theScript`". These operators are represented by instances of classes that extend `DefnFunc`.
`AgentList`	`usersList`	A list of user information for user names and passwords, enabling multi-user support for script notations. This list contains all users who could possibly interact with a script at some point.
`AgentList`	`currentUsers`	This list contains users who are currently registered as *logged in* and hence are able to make new definitions and redefine existing definitions (subject to currently set permission values).

Table 6.4: Data fields (all are `private`) for a `JaM.Script` class.

dependency between definitions on the queue and script and for type clashes caused by implicit definitions in the queue.

Then the process for adding to the queue and then updating continues in a continuous loop, forming the interaction procedure for incremental update of the JaM script. When an update occurs, the definitions in the queue are checked and then used to modify the current state of script S. With reference to DM Model $\mathcal{M}_S = (A, F, D, V)$, the role of the set of references A is played by the "`theScript`" list of definitions. For each definition reference in this list: the "`f`" field represents the mapping from the reference to the operator, in a similar way to the DM Model mapping F; the "`dependents`" field represents the list of arguments of the operator, in a similar way to the DM Model mapping D; the "`value`" field represents the current value, in a similar way to the DM Model mapping V. The counterpart of the set K of redefinitions is the "`queue`" field in a JaM script class. The algorithmic procedure to update the current state of a JaM script is based on the block redefinition algorithm.

The most important methods that can be called by a programmer to interact with an instance of a `JaM.Script` class are shown in Table 6.5[4]. The table shows the name of a method, the types of the parameters that it is passed, the Java type of the value that the method returns and a description of the method. Several of the methods are passed an integer *cookie* value (see page 202 for the definition of a *cookie*) before other parameters. Where this is the case, this is a unique identifier for a user who is currently logged into the script and has requested the action prescribed by a call to the associated method. This allows the method to throw an exception (`JAMException`) if the user does not have permission to carry out the action. Multi-user support is described further in Section 6.3.

Once a new instance of the `JaM.Script` class is created for a particular JaM

[4]This is apart from the constructor method for a `JaM.Script` that is described in Table 6.7.

Method	**Passed**	**Returns**	**Description**
`addToQ`	`int`, `String`	`void`	This method is called with a `String` representation of a JaM script redefinition and adds this to the `queue` of redefinitions.
`update`	`int`	`void`	This method is called to update the main script (`theScript`) consistent with the redefinitions on the `queue` of pending definitions.
`addType`	`int`, `DefnType`	`void`	This method is used to add a data type to the `tlist` of an instance of a `JaM.Script` for use in definitions of `theScript` to create the JaM notation data types for the script. Though this method is normally called during the initialization of a script, it can be called at any time during the life of a script instance, dynamically extending the type structure of the JaM notation.
`addFunc`	`int`, `DefnFunc`	`void`	This method is used to add an operator to the `flist` of an instance of a `JaM.Script` to create the underlying algebra for the script. This method is normally called during the initialization of a script, but can be called at a later point to dynamically extend operators available in the underlying algebra of a JaM notation.
`printVal`	`int`, `String`	`String`	Convert the current value of the *identifier* with the same name as the passed `String` into a string of characters in the format "*identifier* = *value*".
`printDef`	`int`, `String`	`String`	Convert the current state of the implicit definition associated with the *identifier* name passed as a `String` into a string of characters in the format "*identifier* = *operator* (*arguments*)". If the definition is explicit, the behaviour of this method is the same as `printVal()`.
`printAll`	`int`	`String`	This method converts every definition in a script into a string containing a JaM script for all the definitions. If the definition is implicit then the implicit format is used as for `printDef` method, otherwise if it is explicit then the explicit format is used as for the `printVal` method.

Table 6.5: Methods available for a programmer to communicate with an instance of a `JaM.Script` class.

notation, the data types and operators of that notation should be added to the instance to form the `tlist` and `flist` data fields. For each data type Y, a new instance of the class that represents Y should be created and this instance used in an argument to the `addType()` method[5]. The order in which this method is called is important as it affects the way in which data types are recognised in explicit definitions. The first data type added to the script is the last data type that a value string will be checked against for a true value from its `recognise()` method and vice versa. The process of calling the `addType()` method is similar to the need to carefully order the regular expressions in the input file to a scanner tool, such as "`lex`" [LMB92].

An example of this is shown in the code in Appendix A.3 where the call to `addType()` for the integer data type occurs after the call of the method for the floating point data type. This is so that numerical value strings that do not contain a period (full stop) character (or an "`e`" or an "`E`"), are recognised first as integers. If the string does contain a period it is not then recognised as an integer but as a floating point value.

Operators for the underlying algebra over the data types are added by calls to the `addFunc()` method. For this purpose, a new instance of the class representing the JaM notation operator should be created. The string name given to the instance should be the same as the name that will be used for the operator in the JaM notation. The `flist` field of an instance of a `JaM.Script` class is updated using the `addFunc()` method. The order in which the operators are added is not important[6].

[5] Only the `root` user can call the `addType()` and `addFunc()` methods. See Section 6.3 for more information.

[6] This is true as long as every operator has a different string name. No automatic check is in place to determine the uniqueness of these names.

6.2.5 Explicit Arguments to Implicit Definitions

One built-in feature of the JaM Machine API to assist a user interacting with a script is the ability to use an explicit value as an argument to an implicit definition. Consider the example where the `addToQ()` method of the `JaM.Script` class is called to introduce an implicit definition of the form "*id1* = *operator* (*arg1*, ..., *arg4*)". If the string of characters for *arg2* and *arg4* are explicit values instead of other identifiers that already exist in the script, then additional definitions are automatically created for these values. In the example below, the implicit definition is replaced by three definitions, two explicit and one implicit:

id1_2 = *arg2*
id1_4 = *arg4*
id1 = *operator* (*arg1*, *id1_2*, *arg3*, *id1_4*)

6.3 Multi-user Environments for JaM Scripts

A JaM script represents a set of definitions for values and persistent indivisible relationships between these values. These scripts are textual entities that can be shared between users by agreement that "*this script is the current state of our model*". Empirical modelling principles have been proposed as a good way in which to implement shared modelling systems to support methodologies such as concurrent engineering [BACY94b]. The JaM Machine API was the first tool to offer some technical support to a truly multi-user process for the creation and interaction with definitive scripts. If several different users are contributing to the definitions of a script simultaneously then it is useful to record who contributed which definition. By recording this information in instances of the `JaM.Definition` class, it is possible to ascribe ownership to each definition and manipulate this ownership so as to protect users who do not own a definition from redefining it.

Section 6.3.1 discusses the concept of permissions for controlling which definitions can be manipulated by which users, known as *definition permissions*. The mechanism used for controlling access to definitions is similar to that for controlling access to files on a multi-user computer operating system. The concept of agency used in the JaM Machine API is discussed in Section 6.3.2. In the following section, this is compared to the LSD notation [Bey86b], which is used for the analysis of agency in the modelling real-world situations and systems. The API can support some parts of the LSD notation in the implementation of models that use JaM notations.

6.3.1 Definition Permissions

One motivation for a modeller to construct a computer-based model of a real-world situation is as a mechanism to communicate qualities of the real-world referent through the model to other people. Another motivation is to assist in the process of thinking and reasoning about some real-world referent. In sharing a model, a modeller may wish to ensure that another user cannot undermine the integrity of the model. For example, in a model of a radio a modeller way not wish a user to be able to redefine the function of a control knob for tuning in stations so that it alters the volume of the sound instead. The modeller may also wish to protect definitions from accidental modification by themselves, as they are now committed to an explicit value or implicit indivisible relationship as observed from the referent.

For large models or models that require the skill specialisation of more than one person in their construction, a desirable environment is one which supports collaborative construction of models by more than one modeller simultaneously. In such an environment, modellers need to be able to protect their parts of the model to prevent accidental modification by users who do not comprehend their definitions and share other parts of the model with other modellers, either to allow them to reference them in their own work or to allow them to interactively modify them.

For example, in the design of a new transistor radio, an electrical engineer designing the layout of a printed circuit board may wish to protect their layout of components from modification by the designer of the external case, but allow this designer to reference the current location of certain components on the board such as the volume control. The stylistic designer of the case may have control over the overall internal geometry of the case to which the electrical engineer must conform. Both modellers may have the ability to decide where to locate the headphone socket. All conflicts of interest need to be settled by negotiation between the two modellers.

When models are constructed as scripts of definitions, the text for these definitions can be easily shared between modellers through any means of textual communication. It is possible to associate each definition with a modeller by annotating the text. The scenarios discussed above can be classified as:

- modeller and user;
- modeller and commitment to components of a model;
- teams of collaborating modellers.

By associating with each definition information describing who has the ability to modify (redefine) or reference (request the value of) the definition, the scenarios listed above can be realised. The information associated with each definition is called the definition permission and, if implemented in a tool for dependency maintenance, can be used to control simultaneous access and interaction with one definitive script by several modellers and users simultaneously.

The concept of permissions associated with definitions in the JaM Machine API is based on an existing and proven method for users to share work and information while keeping some information private and personal. Multi-user computer operating systems, such as UNIX [Joy94] and Microsoft's Windows NT [CS98], require a user to identify themselves prior to the start of an interactive session by typing a system-wide public username and secret password. During this session, all files which the user modifies and processes that they run are associated with their owning user. It is the job of users to make sure that they do not make available to others any files that are of a private nature and make public any information that they would like to share. It is also up to the user to protect important files so that they do not accidentally delete them or modify them once they are considered final. In this way, several users can interactively share the same devices such as disks, printers, processors or memory. Overriding control of the permissions to use a resource is given to a super-user, who is called "`root`" on UNIX or the "`Administrator`" on Windows NT.

On multi-user, computer-based filing systems, data files are protected on a file-by-file basis. To achieve the file protection on UNIX, each file has an associated *inode* that contains information such as date and time of the last modification of a file, the locations of a file on the physical disk surface and so on. It also contains some permission fields and owner information that describe information about which user owns the file, which group of users owns the file, whether the owning user can read, write/modify or execute the file, whether members of the owning group can read, write or execute the file and whether any other users can read, write or execute the file. Information similar to that recorded in an inode is associated with each definition by the JaM Machine API. In the current version, there is no support for groups and no concept such as the *execution* of definitions. For every definition, there is both an associated owner and permissions. The data fields in a `JaM.Definition` class are shown in Table 6.1.

JaM Machine permissions can be expressed as a string made up of four characters, of the form "`rwrw`". Each character is either as shown in the quotation marks to indicate that the permission associated with the character is set, or a dash "`-`" to indicate that it is not set. For example, the first character is an "`r`" if the owning user has the right to reference the definition in the argument list to another definition, or can inspect the current value or implicit definition[7] in the current state of the script for the definition. If the user does not have these permissions, the first character is set to "`-`".

The second character for a definition permission refers to the owner's permission to be able to redefine the associated definition, explicitly or implicitly. It is a "`w`" if they can and a "`-`" if they cannot. The final two characters are the same as the first two except that they concern the permission for all non-owning users of a definition in a script to interact with the associated definition. The ways in which definition-owning agents can interact with scripts containing definitions are described in the next section. The default definition permissions set for a new definition in a script is owner reference and modify and non-owner reference only ("`rwr-`").

6.3.2 Definition-Owning Agents

Each potential definition owner is considered as an agent who can interact with a JaM script, contributing redefinitions. Every agent has a username and a password, the username uniquely identifying the agent. In the JaM package, agents are represented by instances of the class `JaM.Agent`. Every JaM script has a user list (`usersList`) of agents who are permitted to join interaction with it and a list of current users (`currentUsers`) that are actively interacting with the script (see Table 6.4).

To create a multi-user environment, a special super-user agent is required

[7]To inspect the current value of a definition, the `printVal()` method of `JaM.Script` is used. To inspect the current implicit definition, the `printDef()` method is used. See Table 6.5.

with the ability to add and delete other agents from the overall list of users. This super-agent is called "`root`" and every script must have one. When a script is initialized without an agent list specified, the only user is the `root` user. If a programmer wishes their script to only be interacted with by one agent at a time then this one agent should have username "`root`". This user still has to become active by *logging in* before they can call any of the other methods for an instance of a `JaM.Script` class.

Table 6.6 shows the methods in the `JaM.Script` class that support multi-user interaction. To create a new JaM script, a programmer first has to call the constructor method `Script`. Two variations to this method are available, depending on the types of the arguments that it is passed. To start a new agent database with only the `root` user or to initialize a script with only one agent, the method is called with an empty argument list. This generates a new instance of a script and the `root` agent must then use the `login()` method to start using it. This process assigns the `root` agent as currently active and generates a random and unique integer *cookie* that is used in other method calls of the script instance attributed to `root` until the agent logs out.

The code excerpt shown below demonstrates the single-user mechanism for the use of an instance of the `JaM.Script` class. The first two lines of the example code below shows the minimum code required to create a JaM script `s1`. The following lines contain code to add one data type and operator, add a new definition to the queue and update the current state of the initially empty script. When the `root` agent has finished interacting with the script, it is *logged out* using the `logout()` method.

```
Script s1 = new Script();
int root = s1.login("root", "PHDBOD");
⋮
s1.addType(root, new JaMInteger("JaMInteger"));
s1.addFunc(root, new JaMAdd("add"));
s1.addToQ(root, "a = 34");
s1.update(root);
⋮
s1.logout(root);
```

Notice from the code above how the `root` agent's *cookie* must be passed to all method calls to associate the method call with the `root` agent. In the example code, the identifier "`a`" is set to have owning agent `root`. At the end of the example, `root` is logged out using the `logout()` method and is no longer interacting with the script. The next time that the `root` agent logs on, they will be assigned a different *cookie* for reasons of security. It is by passing the *cookie* every time that a method such as `addToQ()` is executed that allows the system to check whether a particular user agent has permission to carry out a particular action, depending on currently set definition permissions.

It is up to the programmer who writes an application that uses a JaM script to prescribe how agents are going to interact with a script. It is possible to create completely computer-based autonomous agents, perhaps executing in different threads, that perform some regular or randomly-occurring redefinition. To protect the integrity of the redefinitions of an autonomous agent, any definitions that the autonomous agent shares should be associated with the definition permissions "`rwr-`". Other user agents may be connections to computer terminals with textual interfaces directly to the script and other agents may be interacting with the script through a prescribed graphical interface with buttons, tick boxes, choice menus and so on. The programmer of an application needs to ensure that all users log in and that every action that the agents subsequently takes results in a method call to the script with the agent's *cookie* as the first argument.

The information about agent owners for definitions and current definition permissions is not recorded in the textual version of a JaM script. This information must be requested using the `inspectDef()` method of the `JaM.Script` class or can be inferred by the exceptions thrown by calls to methods of this class (see Appendix A.2). A database of usernames and passwords that can be used to instantiate new objects of the `JaM.Agent` class can be saved to disk by the `root` agent using the `savePasswords()` method (see Table 6.7) and loaded in for use in new script instances using the `Script` constructor that is passed a filename (see Table 6.6). Agents can be added and deleted from a script instance by the `root` agent only, using the `addUser()` and `delUser()` methods.

Ordinary agents other than the `root` agent have methods that when called with their *cookie* allow the modification of passwords (`password`) and definition permissions (`permissions`), as shown in Table 6.6. An agent can change their own password using the `password()` method of an instance of a `JaM.Script` class by passing their current *cookie* and password followed by a new password repeated twice[8]. An agent can change the definition permissions of a definition that they own using the `permissions()` method that alters the permission fields inside an instance of a `JaM.Definition` class, which are shown in Table 6.1.

6.3.3 Agents in the JaM Machine API and the LSD Notation

The LSD notation [Bey86b] is a specification notation for the description of agency in models of systems that are concurrent and/or reactive. The notation is not directly executable and describes agents in terms of: identifiers in a definitive script that are associated with their current state (*state variables*); state variables for other

[8]The password argument is repeated in this method call to encourage good practice in user-interface design by programmers.

Method	Passed	Returns	Description
`Script`	`String`	`void`	Constructor method for a new instance of a `JaM.Script` class. The `String` argument should represent the pathname to a file containing a user database for the new instance of the script class.
`Script`	—	`Script`	Constructor method for a new instance of a `JaM.Script` class. The new instance will have one super-user called "`root`" with a password "`PHDBOD`".
`login`	`String`, `String`	`int`	This method is passed a username and password and, if the user exists in the password database (`usersList`), returns a *cookie* integer that the user can then use to perform all their interaction with the script. The user is added to the list of logged in users (`currentUsers`).
`logout`	`int`	`void`	Log the user with the passed integer *cookie* value out from being able to interact with the instance of a script. The user is removed from the list of currently logged in users (`currentUsers`).
`addUser`	`int`, `String`, `String`	`void`	Add a user to the list of users who can interact with the instance of a script (`usersList`). Only the `root` user can do this and must provide their current *cookie*, a username for the new user and and initial password for the new user.
`delUser`	`int`, `String`	`void`	Remove a user from the list of users who can interact with this instance of a script (`usersList`). Only the `root` user can do this and must provide their current *cookie*, and the username for the user to delete.
`permissions`	`int`, `String`, `boolean`, `boolean`, `boolean`, `boolean`	`void`	Method by which the owner of a definition can change its current permission values. A user can only change the permission of a definition that they own. The first argument is the user's current *cookie*, followed by the name of the definition, then four boolean values corresponding to `ownerRead`, `ownerWrite`, `allRead` and `allWrite` fields respectively (see Table 6.1).
`password`	`int`, `String`, `String`, `String`	`void`	Method for a user to change their own password. The first argument is the users current *cookie*, the second their current password and the third and fourth should be the new password repeated twice.

Table 6.6: Methods for the `JaM.Script` class for controlling multi-user interaction with an instance of the class.

Method	Passed	Returns	Description
`inspectDef`	`int`, `String`	`String`	Format internal information about an instance of a `JaM.Definition` class and return it as a string. Returned information is "*identifier owner permissions* `->` *dependents*".
`savePasswords`	`String`	void	Save current username and password database for an instance of a `JaM.Script` to an encoded file. This file can be subsequently reloaded using the `Script` constructor method.

Table 6.7: Utility methods in the `JaM.Script` class.

agents that they can observe (*oracles*); state variables for other agents that they can redefine (*handles*); definitions derived implicitly from their current state and their oracles (*derivates*); guarded redefinition actions that represent the agent's protocols to take action based on their current state and derivates (*protocol*). An LSD specification cannot be interpreted operationally without providing additional runtime information, such as priorities for the order in which the guards for redefinitions are examined and, if true, the order that the a guarded actions (redefinitions) are executed. Ness discusses the significance of LSD analysis in connection with software development in detail in his thesis [Nes97][9]

The agency and definition permissions in the JaM Machine API can be used to support agency in models specified in the LSD notation. In this section, one possible strategy for the interpretation of an LSD specification in a JaM script is presented. The list below takes each component (state, oracle, handle) of an LSD notation for any agent **A** in a specification. Suggestions are given in this list for ways in which a programmer can handle the agent specification when implementing it with the JaM Machine API. The first step is that the programmer should create

[9]Recent and yet-to-be published work on the implementation of an *LSD Engine* was carried out by Alexander Rikhlinsky at the Moscow Engineering Physics Institute as part his MSc. thesis. This work was supervised by Adzhiev.

an instance of a `JaM.Script` with usernames and passwords for agents with the same names as used in the LSD specification.

`state` All state variables for **A** in the LSD specification should be owned by agent `A` in the internal representation of the JaM script. The permissions for these state variables should be initially set to owner reference and modify "`rw--`".

`oracle` All oracles for **A** that are state variables for another agent should have non-owner reference permission initially set to true "`rwr-`".

`handle` All handles for **A** that are state variables of another agent should have non-owner modify permissions initially set to true "`rw-w`".

`derivate` Derivates for agent **A** are implicit or explicit definitions in a script that are either state variables for the agent, handles for state variables of another agent or an internal definition private to the agent. In the case that the definition is internal and private, it should have its non-owner reference/modify permissions set to false "`rw--`". Once derivates are introduced and committed as an accurate description of the observed behaviour of the real agent, their definition permission can be set to disallow any further modification "`r---`".

`protocol` The implementation of the protocol is up to the programmer of the application. One possibility is that each guard is presented to a human user who is representing the interaction of **A** with the model when it evaluates to true and the user takes responsibility for determining the order of execution of the associated redefinition action, or whether the action should be carried out at all! Alternatively, an autonomous agent running as a thread can be introduced to periodically sample the values of guards and make the associated redefinition actions when the guard is true.

In this way, a programmer can use the multi-user support in JaM to assist in some aspects of implementing models based on LSD specification by making sure that the definition permissions are set as strictly as is possible, to meet the conditions of the list above. However, once an identifier in a script is an oracle or a handle for one agent other than the owner, it is open for redefinition by any other non-owner agent and does not provide protection of the definition consistent with the LSD specification.

By way of illustration, consider the excerpt of an LSD specification describing the second-hand and minute-hand agents of a clock given in Figure 6.3a. Other agents exist in the clock specification, such as an oscillator that increments the state variable `seconds` of the second-hand agent. An interpretation of this specification as a JaM script[10] is shown in Figure 6.3b, in which the definitions are split between the agent owners `second_hand` and `minute_hand`. The permissions for these definitions are shown in the right-hand column of this table. Notice how the derivates are kept private to the owning agent. Figure 6.3c shows some Java code for the `run()` method of a Java thread that could be used to implement the `protocol` section of the second-hand agent, with a script instance called `s1` and an integer *cookie* variable for the currently active second-hand agent identified as `second_hand`[11].

The `minutes` identifier in Figure 6.3b has non-owner modify permission so that the second-hand agent can update the value through its protocol. Nothing now prevents another agent other than the `second_hand` agent from modifying the values of minutes which may or may not be part of the whole LSD specification. The programmer should provide a level of protection for such a definition in their own code if they wish to enforce strict adherence by uses to the specification.

[10]The implicit definitions using the `eval` operator take the first argument as a description of a template expression tree and substitutes the second argument as `$1`, the third argument as `$2` etc., prior to evaluation.

[11]The thread sleeps for 0.1 seconds before performing its next check of the guards.

```
agent second_hand() {
state
  seconds sec_length
  sec_angle xsec ysec
handle
  minutes
derivate
  sec_angle = (pi / 2)
    - (pi * seconds) / 30,
  xsec = sec_length * sin(sec_angle),
  ysec = sec_length * cos(sec_angle)
protocol
  seconds == 60 ->
    minutes = minutes + 1,
  seconds > 60 ->
    seconds = 1
}
```

```
agent minute_hand() {
state
  minutes min_length
  min_angle xmin ymin
oracle
  sec_length
handle
  hours
derivate
  min_length = 0.75 * sec_length,
  min_angle = (pi / 2)
    - (pi * seconds) / 30,
  xmin = min_length * sin(min_angle),
  ymin = min_length * cos(min_angle)
protocol
  (minutes == 60) ->
    hours = hours + 1,
  (minutes > 60) ->
    minutes = 1
}
```

a

Owner	Script	Permissions
second_hand	`seconds = ... // Handle for oscillator agent`	r-rw
	`sec_length = ... // Handle for designer agent`	r-rw
	`sec_angle = eval("(pi / 2) - (pi * $1)/30", seconds)`	rw--
	`xsec = eval("$1 * sin($2)", sec_length, sec_angle)`	rw--
	`ysec = eval("$1 * cos($2)", sec_length, sec_angle)`	rw--
minute_hand	`minutes = ... // Handle for second_hand agent`	r-rw
	`min_length = eval(0.75 * $1, sec_length)`	rw--
	`min_angle = eval("(pi / 2) - (pi * $1)/30", minutes)`	rw--
	`xmin = eval("$1 * sin($2)", min_length, min_angle)`	rw--
	`ymin = eval("$1 * cos($2)", min_length, min_angle)`	rw--

b

```
void run {
  String smin;
  while (true) {
    synchronized(s1) { // get a lock on object s1
    if (s1.printVal(second_hand, seconds).endsWith("60"))
      {                                  // check first guard
        smin = s1.printVal(second_hand, minutes).lastToken();
        s1.addToQ("minutes = " + (Integer.valueOf(smin) + 1));
      }                     // add redefinition to queue if true
  smin = s1.printVal(second_hand, seconds).lastToken();
  if (Integer.valueOf(smin) > 60)        // check second guard
    s1.addToQ("seconds = 1");            // add redef. if true
  s1.update(second_hand);  } // update s1 and release lock
  Thread.sleep(100);
  }
}
```

c

Figure 6.3: LSD specification with a JaM script representation and thread implementation of a `protocol` section.

6.3.4 Extension of Agency in JaM Scripts

In this section of the chapter, a mechanism for associating ownership and permissions to reference/modify definitions has been discussed. This mechanism is limited to associating one owning agent to each definition and four permission bits, but is sufficient for proof of concept. The data fields of the `JaM.Definition` (Table 6.1) and `JaM.Agent` classes can easily be extended to include additional information such as group ownership and permissions for definitions, date and time of last modification, size of the definition in memory, and any other useful definition management information similar to that included in a files *inode* on a UNIX system. The methods of the `JaM.Script` class can also be extended to increase the flexibility of the management of interacting users in a multi-user script environment, via methods to control group access and the transfer of ownership of definitions. In the future, support should be provided for representing hierarchies of definitions in directories of a filing system or tables of a database [Bow93].

6.4 Implementation Mechanisms for JaM Scripts

Many implementation mechanisms for applications that support or integrate JaM scripts are available. These include:

- the standard single-user mechanism (as illustrated on page 202);
- the dynamic extension of the data types and operators of a JaM script on-the-fly during execution of the script interpreter, where the data type and operator structure is not fixed at any point. This model is described in Section 6.4.1.
- distributing interfaces to scripts across several workstations with client/server systems. This is described in Section 6.4.2.

- using scripts that include data types for other scripts. This mechanism is known as the *scripts within scripts* mechanism and is presented in Section 6.4.3.

6.4.1 Dynamic Extension of JaM Notations On-the-fly

A JaM script offers the same level of open-ended functionality to a user as the EDEN interpreter in the manipulation of definitions in scripts but does not allow for the redefinition of operators on-the-fly as is possible in EDEN. The data types and operators in JaM scripts are implementing by a programmer by extending some classes of the JaM Machine API. These are added to the notation during execution of one of the implementation mechanisms, which allows the notation to be different for specific applications. Hence the types and operators of a JaM script recognised by the API's parser are not rigidly defined prior to the initialization of a JaM script. This allows a user who is also a competent Java programmer to introduce new data types and operators over those data types during interaction with a script. This provides the open-ended functionality of the EDEN interpreter to extend and redefine operators, albeit in a comparatively clumsy manner[12]. The mechanism also extends EDEN's functionality as new explicit data types can be introduced on-the-fly.

The *dynamic data type and operator extension* model for implementing applications based on JaM scripts involves a modification to step 8 of the standard model. The implementation can continue to call `addToQ()` and `update()` repeatedly to create new definitions and redefine existing definitions. At any point in this process new types and operators can be added by following the step-by-step process.

1. Create a new instance of the `JaM.LoadJaMClasses` class. Within this list, it assumed that this instance is identified as "`jcl`".

[12]The classes must be compiled into Java bytecode before being loaded into the running Java Virtual Machine and then added to the type of operator list of a JaM script instance.

2. To load a data type class that extends `DefnType` or an operator type that extends `DefnFunc`, call the method "`jcl.loadClass("`*full_classname*`")`", which returns an object `c` of type `java.lang.Class`.

3. Cast a new instance of this class to `DefnType` if it represents a data type and to a `DefnFunc` if it represents an operator to be added to the current script instance, eg. "`DefnType dt = (DefnType) c.newInstance();`".

4. If the loaded class `c` represents a new data type, call the `addType()` method for the script instance. If the loaded class `c` represents a new operator for the script, call the `addFunc()` method for the script instance.

5. The newly loaded data type or operator is now available for use in calls to the `addToQ()` method in the dynamically extended JaM notation. Newly compiled classes may require significant testing and may generate runtime errors.

The JaM programmer can build up libraries of classes that they wish to use in several different applications. By using the dynamic extension of scripts, a programmer can create an environment where a user can experiment with different data types and operators. In a geometric design application such as the *empirical world builder* presented in Chapter 8, the dynamic extension model can be used to introduce new classes of shape types during a modelling session without the need to stop, link the object code for the application and restart.

6.4.2 Multi-user Client/Server Implementation Mechanisms

For agents to collaborate simultaneously in the interactive construction and experimentation with a JaM script as described in Section 6.3, it is necessary to provide separate interfaces for each agent. The core Java programming API provides several mechanisms to enable a programmer to construct multi-user applications. This includes the creation of threads to handle concurrent components of processes and to take advantage of multi-processor parallel computer systems. Support for TCP/IP networking is available within classes that implement servers to listen and accept network connections, and clients that open communication sockets to these servers. Network support for modelling with JaM scripts is the topic in this section.

The distribution of interfaces to JaM scripts over networks with server/client implementation mechanisms provides the basis for a programmer to construct applications such as multi-user spreadsheets (cf [Nar93]) and tools to support concurrent engineering. The users of these applications can share a workstation at different times of the day, cooperate in an interactive session in the same office or potentially anywhere on an interconnected TCP/IP network, including over the Internet [Kro94].

A *JaM client* is defined as an interface through which an agent can perform redefinitions for a JaM script. This interface is not necessarily textual, it could be a graphical user interface or a connection to a real-world sensor such as a digital thermometer. A *JaM server* is a Java application containing at least one instance of a JaM script class. The standard implementation mechanism, where only the `root` agent interacts through a single JaM client with a script is the simplest of all client/server models. Methods of classes in an application based on the standard model can call the methods of the script instance and pass the root agent *cookie* (see page 202) to every method call. The JaM client in this case can either be an object in the current implementation (such as a text area) or in a separate general application such as a `telnet` client connected to a JaM server socket network port, through which the server provides interaction with its script.

When there are several concurrent agents, many configurations that can be considered for the connection of JaM clients and JaM servers. Figure 6.4 shows three possible configurations. In this figure, boxes with a thick border represent

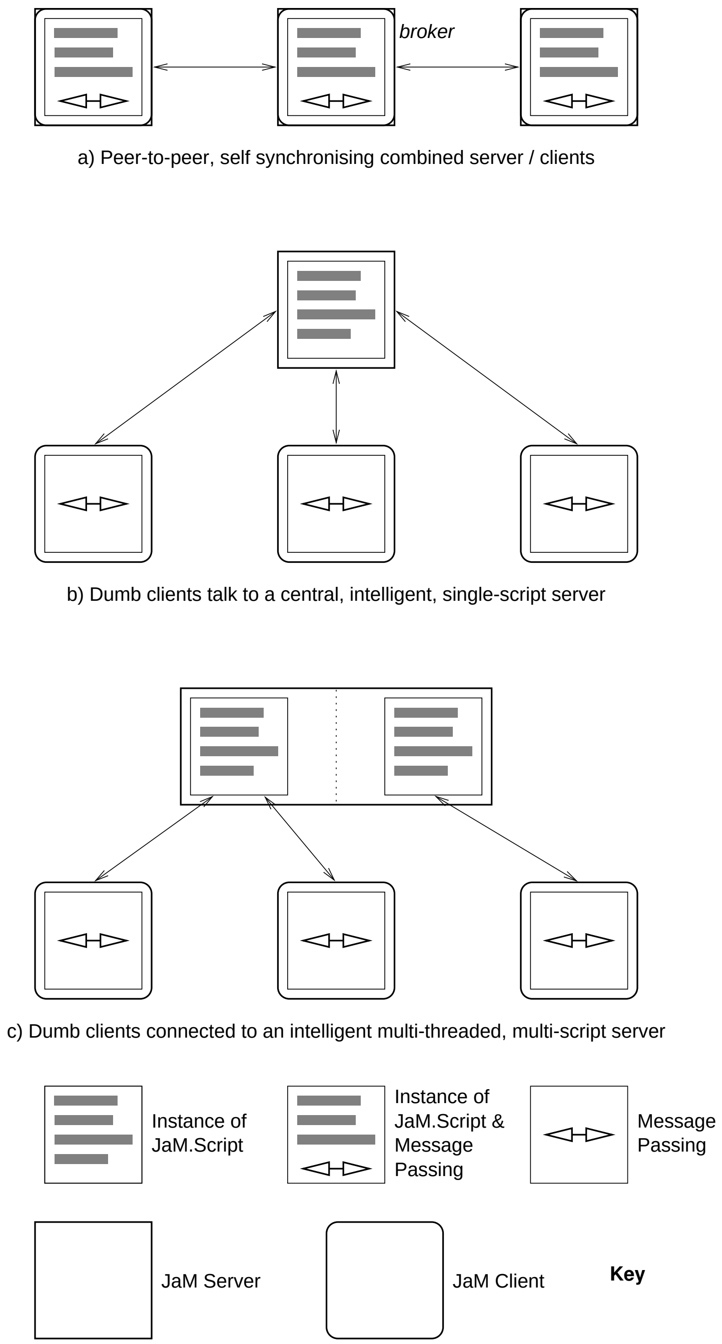


Figure 6.4: Client/server implementation models for multi-user JaM scripts.

a process running on an independent Java virtual machine, either on the same or different computer systems. The thinly bordered boxes represent either JaM script instances or implementation of message passing mechanisms over network sockets for definitions and interactive communication between processes. The figure has three sections 6.4a, 6.4b and 6.4c for three models, with a key shown beneath. Each model is described in the list below.

6.4a - Each JaM client has its own combined JaM server and therefore its own script instance in this *peer-to-peer* model. The programmer must implement a synchronisation mechanism to ensure that any definition that exists in the script of the JaM client of one agent is also propagated to the other agents. Similarly, the current *logged-in* status of agents must be distributed across all scripts and that each script is for the same JaM notation. One client can exists with a greater degree of *intelligence* than the others. This can act as a broker for the synchronisation information and as a super-user agent with higher-level privileges to modify scripts on the other clients than available at the clients themselves.

6.4b - A dumb client is a JaM client in an application that has no associated script instance as part of its code. In this example, three interface clients are connected to a JaM server process that can handle the interaction of several agents simultaneously interacting with the same JaM script instance.

6.4c - Multi-threaded JaM servers can contain more than one JaM script instance and provide a way for agents using dumb JaM clients to choose whether to start by instantiating a new script or to join in interaction with an existing script. The JaM server may also offer a user agent more than one JaM notation that they can instantiate a JaM script object for and then use interactively. The figure shows two agents at dumb JaM clients interacting with the same script and another agent interacting with a separate script independently.

In each of the example models, the programmer needs to make sure they handle the ordering of calls to the `addToQ()` methods and the subsequent `update()` method to ensure that the behaviour of the script is synchronised and predictable[13]. It is also important that a programmer ensures that any exceptions thrown by new definitions on the queue of definitions, such as reporting typographic errors in a definition, or exceptions thrown by an update such as the detection of a cyclic dependency, are sent to the appropriate agent who caused the exception.

6.4.3 Scripts Within Scripts

It is possible to have an instance of a `JaM.Script` class as a data field in a class that extends `JaM.DefnType`. In other words, it is possible to create a data type in a JaM notation for the representation of JaM scripts. Every script has a textual description and so is an appropriate data type to use in a JaM notation. It is also possible to define operators that maintain implicit dependencies between scripts. For example, consider a notation where a script with identifier c is defined persistently and indivisibly as the concatenation of script a and script b. In this *scripts within scripts* model, it is possible for a programmer to consider ways in which to implement definitive notations that contain the higher-order dependencies introduced in Section 3.2.2. An implicitly defined script need not be in the JaM notation as the script that contains its definition.

In this way it becomes possible to define generic templates for scripts that, with some variable parameters, can be instantiated to create new scripts whose structure depends implicitly on the value of these parameters. This is similar to

[13]An example of how to do this in the thread code shown in Figure 6.3c using the Java `synchronized` statement [Fla96]. The method grabs a lock on the script object `s1` until the update is completed successfully and the lock is released at the end of the block.

the graph construct available in the DoNaLD notation, of which the speedometers shown in Figure 3.5 are an example.

6.5 A Simple Illustrative Example - *Arithmetic Chat*

The case-study presented in this section is for a simple JaM notation with two data types and one operator. The code for the classes that extend `JaM.DefnType` and `JaM.DefnFunc` is presented and explained. These data types are then integrated into an instance of a `JaM.Script` class, the script for which can be shared between several users through a mechanism similar to an interactive Internet chat program, such as *Internet Relay Chat* (IRC) [Kro94]. The two data types are integers and floating-point numbers and the operator is addition. These are to be used in a proof-of-concept application in which agents collaborate and discuss scripts of definitions over these two data types with the one operator.

Data Types

The arithmetic chat application has two data types, one for representing integer values and the other for representing floating point values. To represent these as a JaM script requires two classes that each extend `JaM.DefnType`, one to represent integers called `JaMInteger` and one to represent floats called `JaMFloat`. The first step in the process of creating these data type classes is to decide on an appropriate internal data representation in Java for these data types. For these particular data types this is a trivial process as both have direct representation in Java, the `int` type for integers and the `float` type for floating point numbers.

The next stage is to embed this internal data representation in as the data field(s) in the class that represents that data type. Figure 6.5 shows annotated source code for the `JaMInteger` class with the one data field "`int d`" that will hold the internal data representation for an instance of the class. (The `JaMFloat` source code is very similar to the `JaMInteger` code and so is omitted). The internal data representation in this class is "`float d`".

The ordering of methods in a class is not significant but the class must contain implementations for all abstract methods of the superclass together with a constructor method. One possible way to proceed with implementing a data type class is described here. Having chosen the internal data representation, a unique integer reference number for the data type should be chosen and the constructor method for the class added as its first method. This should be followed by the `makeNew()` method that follows the standard template described in Table 6.2.

Two methods are required for conversion to and from internal data representations and strings, `printIt()` and `parseIt()` respectively. The `printIt()` method is passed information about the current formatting of the output line on which the string will be printed. These formatting parameters are the current number of spaces inserted as an indent after a new line character, the current position along the line that the string returned will be appended and the length of the current line. In this method, a programmer should ensure that the length of the string describing the internal data representation does not exceed the length of the line, taking account of the current position. If it does then a new line and indent should be inserted. The `parseIt()` method is passed a string that has already been recognised as of the correct type and converts it into the internal data representation. In the example this is done using the Java core API static method `Integer.parseInt()`.

The programmer also needs to implement a method to decide whether a passed string can be converted into the internal data representation (`recognise()`) and a piece of code describing an action (`action`) to be taken every time a value of this type is updated. In the `recognise()` method shown in Figure 6.5, the string passed is checked to see that it is a sequence of digits, preceded by an optional minus

```
class JaMInteger extends DefnType {
  int d;

  JaMInteger (String name) {
    super(name);
    ref = 3;
  }

  DefnType makeNew() {
    return new JaMInteger(name);
  }

  String printIt (int indent, int linePos, intlineLength) {

    String strVal = Integer.toString(d);

    if (strVal.length() > (lineLength - linePos))
      {
        StringBuffer sb = new StringBuffer(strVal);
        insertSpaces(indent, sb);
        strVal = sb.toString();
      }

    return strVal;
  }

  boolean recognise (String parz) {

    boolean checkIt = false;
    try {
      long l = Long.parseLong(parz);
      if ((l < Integer.MIN_VALUE) || (l > Integer.MAX_VALUE))
        checkIt = false;
      else
        checkIt = true;
    }
    catch (NumberFormatException e) { checkIt = false; }

    return checkIt;
  }

  void parseIt(String parz) {

    try {
      d = Integer.parseInt(parz);
    }
    catch (NumberFormatException e) { d = 0; }

  }

  void action() { }

}
```

Internal data representation

Code for class "JaMInteger" that extends JaM.DefnType.

Constructor
Construct a new "JaMInteger".

Pass string name to constructor for JaM.DefnType and set the unique reference number for this type.

makeNew

Every class that extends JaM.DefnType must implement a "makeNew" method of this form.

printIt
Convert the internal representation to a formatted string

Formatting parameters

Convert internal integer "d" to a string.

Insert a new line and some indent spaces if new string will overlap the current line.

recognise
Is the passed string a sequence of "JaMInteger" type?

Convert the passed string to a Java "long". Throws an exception if the string "parz" is not an appropriate format.

Check value in 32-bit int range.

If string "parz" is in range then recognise it. Otherwise do not recognise string "parz".

Number is the wrong format implies string is not recognised.

parseIt
Convert passed string to internal representation

Convert the passed string "parz" into the internal data representation for the class "d".

A default value for "d" incase of internal execution difficulties.

action

Take no action when a value of this type is updated.

Figure 6.5: Annotated code for the `JaMInteger` class for the representation of integer values in JaM script instances.

sign, using the `Long.parseLong()` method which throws an exception if its string argument contains any other alphanumeric characters than those that match with the regular expression for long integers. If the exception is thrown then the method catches it and returns false, in other words the number string was not a sequence of digits and could not be converted into the internal data representation. A check is carried out to see that the number is within the range of an integer before returning true to say that the passed string is recognised.

In this example class, the `action()` has no code body and therefore no effect. The action is intended to be used in other applications by data types such as graphical lines that need to be redrawn every time their value is updated. Once the class is written, it can be compiled and then used in conjunction with other classes of the same package.

Operators

The example application presented has only one operator in its script for defining implicit number values by summing a list of other value. The source code for the methods of class `JaMAdd` for this operator can be easily modified to provide other standard arithmetic operators such as subtraction, multiplication and division. To implement this JaM script operator, a programmer must extend the `DefnFunc` class and implement the code for the body of its abstract methods.

The source code for one possible version of class `JaMAdd` over the two data types represented by `JaMFloat` and `JaMInteger` is shown annotated in Figure 6.6. The class contains a static lookup table for the reference numbers of the types that may be acceptably used as arguments to the operator in a JaM script. This is followed by a constructor method that calls the generic constructor method in the `JaM.DefnFunc` class. No data fields are required in a class that extends `DefnFunc`.

The `typeCheck()` method checks an argument sequence represented as a

```
class JaMAdd extends DefnFunc {

  static int INTEGER = 3;
  static int FLOAT = 4;

  JaMAdd (String name) {
    super(name);
    ref = 1;
  }

  DefnType typeCheck (DefnList dl) {

    DefnType dt = null;
    boolean argsNumeric = true;
    boolean floatFlag = false;

    for ( int u = 1 ; u <= dl.length ; u++ )
      {
        if ((dl.arg(u).ref != INTEGER) ||
            (dl.arg(u).ref != FLOAT))
          { argsNumeric = false; break; }

        if (dl.arg(u).ref == FLOAT) floatFlag = true;
      }

    if (floatFlag && argsNumeric)
      dt = new JaMFloat("implicitFloat");
    if (!floatFlag && argsNumeric)
      dt = new JaMInteger("implicitInteger");

    return dt;
  }

  DefnType f (DefnList dl) {

    DefnType dt;
    boolean floatFlag;

    for ( int u = 1 ; u < dl.length ; u++ )
      if (dl.arg(u).ref == FLOAT)
        { floatFlag = true; break; }

    if (floatFlag == true)
      {
        dt = new JaMFloat("implicitFloat");
        ((JaMFloat) dt).d = 0;
        for ( int v = 1 ; v < dl.length ; v++ )
          {
            if (dl.arg(v).ref == INTEGER)
              ((JaMFloat) dt).d += (float) ((JaMInteger) dl.arg(v)).d;
            else
              ((JaMFloat) dt).d += ((JaMFloat) dl.arg(v)).d;
          }
      }
    else
      {
        dt = new JaMInteger("implicitInteger");
        ((JaMInteger) dt).d = 0;
        for ( int v = 1 ; v < dl.length ; v++ )
          ((JaMInteger) dt).d += ((JaMInteger) dl.arg(v)).d;
      }

    return dt;
  }

  void action (Definition d) {
    System.out.println("Just updated the value of " + d.name);
  }

}
```

Code for class "JaMAdd" that extends JaM.DefnFunc.

Type reference lookup table.

Constructor
Construct a new "JaMAdd".

Pass string name to constructor for JaM.DefnFunc and set an (optional) unique reference number for this operator.

typeCheck
Check types of data in passed argument list to see if they are compatible with this operator.

"dt" represents the type the operator maps to for the given argument list "dl".

Check all arguments to check they are appropriate types for this operator.

Each argument must be either integer or float.

If one argument is a float, the operator maps to a float.

Return "null" if operators are not appropriate, else return either a new float or a new integer.

f
Evaluate the operator for the values in the passed list of arguments.

New object representing left-hand-side value.

Loop to see if return type is float.

Return type is float so create a new instance of "JaMFloat".

Loop evaluates implicit value.

Return type is integer so create a new instance of "JaMInteger".

Loop evaluates implicit value.

Return new implicit value.

action
Called on update.

Debugging action to trace calls to this operator by printing a message of the "standard out" stream.

Figure 6.6: Annotated code for the `JaMAdd` class for summing together a sequence of numerical arguments.

`JaM.DefnList` to see if it contains arguments appropriate to a particular operator. In the example implementation there are only two data types and these are both acceptable arguments to the operator. Future reuse of the operator may require its use in JaM notations with data types such as strings that are not appropriate data types. To allow for reuse, the class has a check built in to check that its arguments are all numerical. No limit is imposed on the length of the sequence of arguments. If one or more of the arguments to the operator is a floating point number then the operator returns a floating point number (a `JaMFloat`), otherwise if all the arguments are integers then the operator returns an integer (a `JaMInteger`).

The method `f()` contains the code that will evaluate the value mapped to by an implicit definition based on this operator. This method requires access to the fields in the objects representing the arguments to the operator. It is necessary to cast these objects to their original type, as specified by their type reference number before the internal values can be retrieved. In the example code in Figure 6.6, if the return type of the operator is a `JaMFloat`, as will have already been established by the `typeCheck()` method, it can be inferred that at least one of the arguments to the operator is a float. To access the internal data values of the arguments to an operator, it is necessary to check whether they each represent integer or floating point values and cast the object appropriately, prior to summing their values. If the type mapped to by the operator is integer then it can be inferred that all the arguments must be integers and so only one cast to the `JaMInteger` is required in the summing procedure.

The `JaMAdd` code in Figure 6.6 includes an `action()` method that is called every time this operator is evaluated. One use for this method is as a debugging tool for a programmer to trace the execution of their implementation from within method calls to the JaM Machine API. This is illustrated in Figure 6.6, where every time the value of an identifier defined by the `JaMAdd` operator is updated, the identifier is printed on the standard output stream. If a programmer requires access to the internal data fields of an instance of a `JaM.Definition` class in their application, a similar mechanism can be used to pass a reference of a definition object processed by JaM into other non JaM Machine API areas of the implementation.

Integration of Scripts into an Application

A proof of concept tool to allow several agent users to interact simultaneously with scripts of definitions for a JaM notation containing floating point and integer data types with one operator add has been constructed. This uses the source code shown in this section in Figures 6.5 and 6.6 to describe the data types and operator for the notation, and is based on the client/server model shown in Figure 6.4b. The source code for the server is shown in Appendix A.3 and the client is a standard `telnet` client program, generally available on operating systems that support the TCP/IP networking protocol.

When the server is run, a new instance of the `JaM.Script` class is created along with a server that listens for connections on a specified port. The implementation adheres to the same procedural order as the standard model described in Section 6.4 until step 6. When the server receives a connection request from a client, it creates a new thread to handle that connection. The user connecting through a client is requested to type their name. This is used as a new username and password for the agent database of the script and the user is automatically logged in. From then on, every line that the user types is echoed to all other concurrent users with their name preceding what they typed. Certain sequences of characters also perform actions.

// definition Add the *definition* to the queue of definitions for the central script and call the update method. If there is an exception, this is reported to all users.

`value` ***identifier*** Request the current value associated with the *identifier* as a string. The value appears in all clients, except when the *identifier* does not exist. In this case, an exception is reported to all users.

`defn` ***identifier*** Request a textual version of the current implicit definition associated with the *identifier*. The value appears in all clients, except when the *identifier* does not exist. If this is the case, an exception is reported to all users.

`?` ***identifier*** Request some additional internal information about the definition associated with the *identifier*, including its current owner, definition permissions and dependencies. This information appears in all clients, except when the *identifier* does not exist. In this case, an exception is reported to all users.

`exit` Close the connection to the client. The server application continues to run.

Once the server is started, user agents can connect. All they need to make a connection is a `telnet` client, specifying the port number to which the server is attached. Figures 6.7 and 6.8 shows an interactive chat session between two users Jane and Mark. Together they construct and inspect a four definition script while holding a textual discussion. Any line of text typed by Jane or Mark is indicated in their respective consoles by a "`=>`" arrow symbol. At the end of the session, Jane tries to redefine the value for `c`, a definition that is owned by Mark, and is refused permission to do so as the default permission set for a new definition is "`rwr-`".

The behaviour of this server is trivial but does demonstrate the potential for sharing JaM scripts amongst clients. Any other JaM data types and operator classes that exist can be instantiated and used in a similar chat application. The `telnet` client can be replaced by a more complex and specialised client that includes

```
=> jane@britten > telnet britten 8861
   Trying 137.205.225.29...
   Connected to britten.
   Escape character is '^]'.
=> Welcome to Arithmetic chat.  Please enter your name > Jane
   Hello Jane, you may now chat!
   Welcome to new user Jane.
   Welcome to new user Mark.
   Mark: Hi Jane! Here is a line of text that will be echoed to you and me.
=> Hello Mark. How are you?
   Jane: Hello Mark.  How are you?
   Mark: Well.  Shall we try some sums ... how about adding 23E3 and 2431.
=> // a = 23E3
   Jane: // a = 23E3
   Mark: // b = 2431
   Mark: // c = add(a, b)
=> value c
   Jane: value c
   script > c = 25431
=> defn c
   Jane: defn c
   script > c = add(a, b)
=> ?b
   Jane: ?b
   script > b Mark rwr- ~> c.
=> How about summing all three values together.
   Jane: How about summing all three values together?
   Mark: // d = add(a, b, c)
   Mark: value d
   script > d = 50862
   Mark: Why don't you redefine the value of c ...
=> // c = -12e-2
   Jane: // c = -12e-2
   Jane: JaMException: addToQ: It is not possible to modify this definition: c
   The modifying agent is not the owner and write protection is set.
=> Oh .. I do not have permission to do this!
   Mark: exit
=> exit
   Jane: exit
   Connection closed by foreign host.
   jane@britten > _
```

Figure 6.7: Jane's console view of an *Arithmetic Chat* with Mark.

```
=> mark@gem >  telnet britten 8861
   Trying 137.205.225.29...
   Connected to britten.
   Escape character is '^]'.
=> Welcome to Arithmetic chat.  Please enter your name > Mark
   Hello Mark, you may now chat!
   Welcome to new user Mark.
=> Hi Jane! Here is a line of text that will be echoed to you and me.
   Mark: Hi Jane! Here is a line of text that will be echoed to you and me.
   Jane: Hello Mark.  How are you?
=> Well.  Shall we try some sums ... how about adding 23E3 and 2431.
   Mark: Well.  Shall we try some sums ... how about adding 23E3 and 2431.
   Jane: // a = 23E3
=> // b = 2431
   Mark: // b = 2431
=> // c = add(a, b)
   Mark: // c = add(a, b)
   Jane: value c
   script > c = 25431
   Jane: defn c
   script > c = add(a, b)
   Jane: ?b
   script > b Mark rwr- ~> c.
   Jane: How about summing all three values together?
=> // d = add(a, b, c)
   Mark: // d = add(a, b, c)
=> value d
   Mark: value d
   script > d = 50862
=> Why don't you redefine the value of c ...
   Mark: Why don't you redefine the value of c ...
   Jane: // c = -12e-2
   Jane: JaMException: addToQ: It is not possible to modify this definition: c
   The modifying agent is not the owner and write protection is set.
=> exit
   Mark: exit
   Connection closed by foreign host.
   mark@gem > _
```

Figure 6.8: Mark's console view of an *Arithmetic Chat* with Jane.

interfaces for graphics and sound which better support interaction with JaM scripts. Because the dependency maintenance is integrated by the JaM concept into a programming API, a programmer can use it in anyway they wish, in combination with any other Java classes and programming APIs.

Chapter 7

Introducing Empirical Worlds

7.1 Introduction

Virtual worlds constructed from three-dimensional geometric objects can be described by the *Virtual Reality Modelling Language* (VRML) [ANM96, SC96]. The language describes shape in a machine-independent way that can then be rendered and explored on a wide variety of computer input and output devices. The language inherits a lot of its design concepts from the *Hypertext Mark-up Language* (HTML) [MK97], which is a machine-independent language for the description of documents containing text, links and images. HTML is viewed on a user's computer screen through a browser or printer where the browser renders the document in a suitable way to present the document to the user on their platform. For VRML, the output devices range from conventional two-dimensional monitors through to immersive virtual reality worlds where a user wears a headset that presents the illusion of walking or flying through a virtual space.

Empirical worlds are virtual reality worlds constructed from scripts of definitions that represent geometric objects and their attributes. Definitions can be used to express observed dependencies between these objects and their defining parameters. *Empirical world* applications can be implemented in Java from a set of *empirical world classes*. They use the JaM Machine API as their underlying dependency maintenance mechanism. An *empirical world* is constructed by attaching an interface to a JaM notation called an *empirical world notation*. This definitive notation is accessed directly by the `JaM.Script.addToQ` method. The right-hand-side of explicit definitions in *empirical world scripts* have similar syntax to VRML nodes.

Empirical world classes can be integrated into applications with support for rendering three-dimensional shapes. The example application in this thesis, called *empirical world builder*, uses a VRML browser as its interface for the interactive inspection of shapes. As definitions in *empirical world* scripts are altered, the corresponding shape in the VRML browser is updated. Many VRML browsers can take advantage of computer-graphics hardware installed on the computer on which they are executing. This hardware is optimised for the display of three-dimensional shape and is to some extent independent of the main processor used for the interpretation and maintenance of dependency for *empirical world* scripts.

VRML scripts are organised with component nodes that can represent primitive shapes, materials, affine transformations, sounds and many other kinds of virtual environment data. The *empirical world* classes demonstrated in this thesis support data types for shapes and shape combinations beyond those that can be described by VRML nodes. Some VRML nodes are used as the basis for the syntax of *empirical world* notations. Not every VRML node has a corresponding *empirical world* class and there are *empirical world* classes that represent geometry that cannot be described in VRML.

Empirical worlds integrate aspects of three different types of modelling, respectively concerned with presentation, description and interaction:

Presentation VRML, including its browser support for the exploration of three-dimensional shapes;

Description the function representation of shape [PASS95];

Interaction empirical modelling using definitive scripts.

In this chapter, *empirical worlds* are introduced by describing the *empirical world* classes and how they achieve this integration. In general, this involves describing how VRML-like syntax is used in explicit definitions for the description of the parametrisations for the mathematical description of point sets using function representation. These include classes that represent *empirical world* notation data types for the primitive shapes (box, sphere, cylinder, cone), affine transformations (scaling, rotation, translation) and binary operations (set-theoretic operations, blending, metamorphosis). In addition to this, the *empirical world* operators for establishing dependency between variables of these data types by implicit definitions are described.

In Chapter 8, applications of *empirical worlds* are discussed. This includes examples of the extension of the classes to include new types and representations for shape primitives and warping transformations. The *empirical world* builder application is described and there is a case-study demonstrating the incremental construction of, and interaction with, a cognitive artefact for geometry.

7.1.1 Motivating Example

As an example of the work presented in this chapter, consider the images of a shape and the associated *empirical world* script shown in Figure 7.1. The *empirical world* script is a description of the shape of a three-dimensional letter **F**. It is parametrised by its `height` and `width`. The shape integrates three shape primitives:

- a cone (`cone1`) that forms the short arm of the letter;
- a cylinder (`cylinder1`) that forms the long arm of the letter;
- a box (`box1`) that forms the spine of the shape.

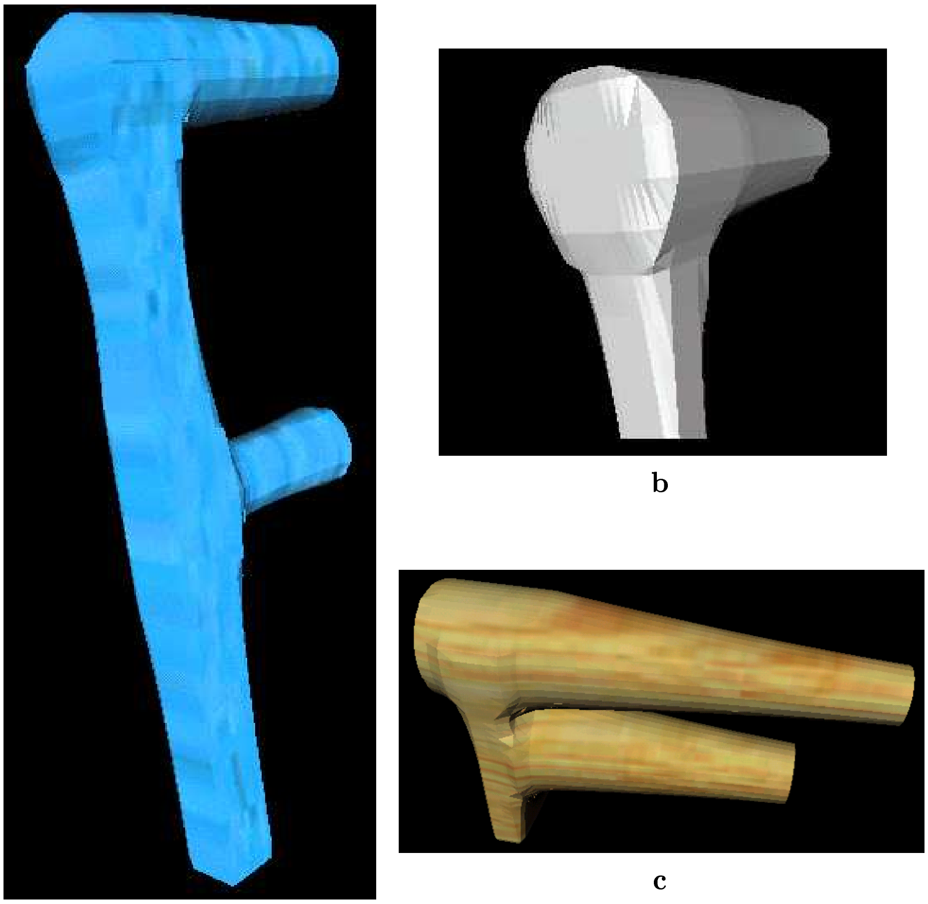
The three primitives are positioned and blended into a union of point sets called `blend1`.

The image of the letter **F** is rendered with a water texture applied in Figures 7.1a and 7.1b. These two images correspond to the script shown beneath the figure. The two images are different views of the same geometric shape, where Figure 7.1b is a *zoomed in* view of the shape to show the detail of the blend between the cylinder and the box. Any definition in the script can be interactively redefined, this will update the representation of the geometry accordingly. Figure 7.1c shows an image of the shape following the redefinition of `height` to a new value of `1.0`[1].

7.2 Empirical World Classes

Empirical worlds can be constructed with the use of *empirical world* scripts that control the instantiation and relationship between objects of *empirical world* classes. These scripts are in a JaM notation that is based on the *empirical world* classes described in this section and Sections 7.3, 7.4, 8.2, 8.3 and Appendix A.4. The majority of the *empirical world* classes are shown in the class diagram of Figure 7.2. The sections of this chapter and Chapter 8 describe an open-ended modelling environment for the creation and exploration of geometric shape on a computer system. The classes are combined in the *empirical world builder*, a server/client definitive environment that supports this open-ended interactivity within a Java applet that

[1]The texture of the image has also been redefined so that a pine texture has been rendered on the surface of the shape.



b

c

a

```
height          = 5.0
width           = 2.0
coneHeight      = multiply(width, 0.6)
coneRadius      = multiply(height, 0.05)
cone1           = cone(coneHeight, coneRadius)
cylinder1       = cylinder(width, coneRadius)
moveCylinder    = multiply(height, 0.5)
moveExtrusions  = multiply(width, -0.4)
transCylinder   = translate(moveCylinder, 0.0, 0.0, cylinder1)
listExtrusions  = list(cone1, transCylinder)
transExtrusions = translate(0.0, moveExtrusions,
                                0.0, listExtrusions)
box1            = box(height, coneRadius, coneRadius)
detail1         = detail 35 25 25
blend1          = blendUnion(detail1, 0.06, 1.0, 1.0,
                                box1, transExtrusions)
water           = ImageTexture { url ".../water.jpg" }
appear1         = appearance( Material { } , water)
attr1           = attribute(appear1, blend1)
```


Figure 7.1: Script and rendering of a letter **F** using *empirical worlds*.

Implicit	`l = 2.0`
	`w = 2.0`
	`h = 2.0`
	`b = box(l, w, h)`

Explicit	`b = Box { size 2.0 2.0 2.0 }`

Table 7.1: Implicit and explicit expressions for a box.

communicates with an embedded VRML browser. This tool is documented in Section 8.4.

In the diagram of classes shown in Figure 7.2, each box corresponds to a class and contains the name of that class. Inheritance between objects is shown from left to right. A class directly connected to and on the right of another class is an extension of the class on the left. Objects shown in boxes with dashed borders are *abstract classes* [CH96]. This means that they cannot be instantiated as new objects during execution as they contain *abstract methods* which have no code in their statement body. These methods must be implemented by all subclasses of the abstract class. All classes in solid boxes can be instantiated as new objects.

The class called `Object` is the `java.lang.Object` that every Java class inherits by default. Inheriting directly from `Object` is the abstract class `JaM.DefnType` that every JaM data type must extend and the abstract class `JaM.DefnFunc` that every JaM operator used for dependency maintenance must extend (see Section 6.2). For most classes that extend `DefnType`, a corresponding class or classes exist that extend `DefnFunc`. These are used to implicitly define variables of the corresponding type in an *empirical world* script. For example, a box centred at the origin with height, width and depth of `2.0` can either be described by a string expression of its explicit value, or by an implicit definition of a box with three floating point values of `2.0`. Table 7.1 shows two expressions for the same geometric shape, where the first is implicitly dependent on the values of `l`, `w` and `h` and the second has an explicit value.

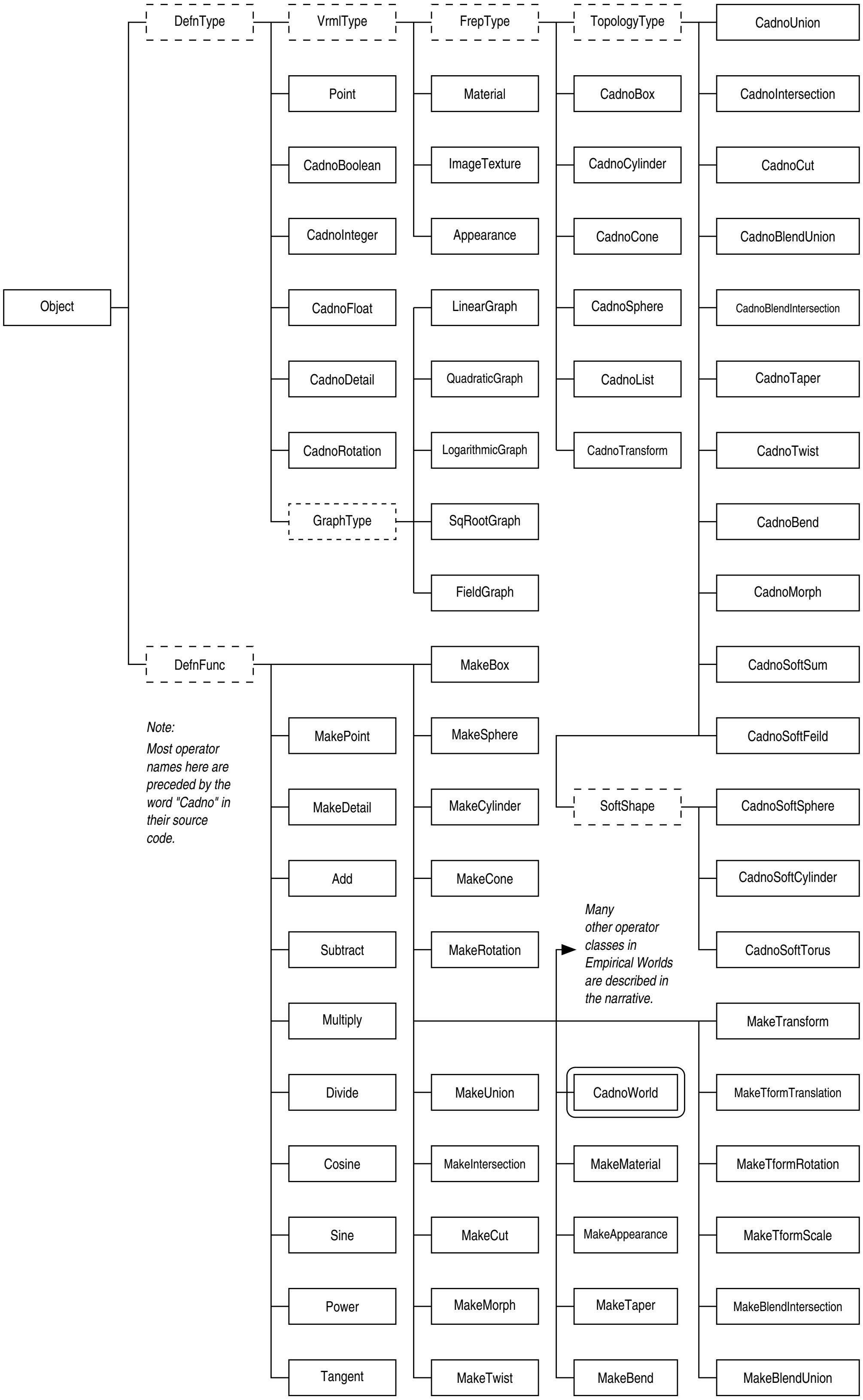


Figure 7.2: Class diagram for the *empirical world* classes.

Classes that directly extend `DefnType` in *empirical worlds* represent data types that have no node to represent them in VRML. These are the basic types of an *empirical world* script, including standard data types for booleans, integers, floating point numbers and three-dimensional points. These are described further in Appendix A.4. Some specialised basic types exists that are specific to *empirical worlds*, including a *rotation type* (`CadnoRotation`) that represents an axis of rotation and an angle of rotation in one data value (see Appendix A.4).

The types `LinearGraph`, `QuadraticGraph`, `LogarithmicGraph`, `SqRootGraph` and `FieldGraph` are classes that extend the abstract class `GraphType`. These types are used to represent single valued continuous functions that map one real number to another. These are required where a definition depends on the shape of a continuous function, such as field functions. They are described further in Appendix A.5. Each function can be sampled at any floating point value x using the built-in operation "`readGraph`" and return a floating point value y that is dependent on x and the graph. The following script will define the value of $y = 2x + 4$:

```
lg = LinearGraph { xCoefficient 2 constant 4 }
y = readGraph(lg, x)
```

Classes that extend `VrmlType`, an abstract class that directly extends `DefnType`, must have a *node* representation in VRML that can be described by a string. For any instance of a class that extends `VrmlType`, the VRML string to describe the instance is generated by a method of the object called `virtualise()`[2]. This method returns a `java.lang.String` object that is a VRML-2 [SC96] node description consistent with the internal data structures of the object instance. Each instance of an *empirical world* class can be described in a VRML-2 file with the string representation generated by this method.

[2]Throughout this chapter, some words in `typewriter` font correspond to identifiers in the *empirical world* classes source code. The name for the `virtualise()` method is chosen to signify that it converts internal object data representations into a representation in the *Virtual* Reality Modelling Language (VRML).

Classes that directly extend `VrmlType` typically represent data types for attributes of worlds, such as materials and textures[3]. For these classes, there is an exact correspondence between the value on the right-hand-side of an explicit definition and the string returned by the `virtualise()` method. For classes that indirectly extend `VrmlType`, if a direct VRML-2 representation is possible then this representation is returned by the `virtualise()` method. If it is necessary to render a more complex shape than one for which a representation exists in VRML-2, a polygonisation or approximation to the represented shape must be provided. This mechanism is encapsulated in classes that extend `FrepType`, as described below.

Any object that extends the class `FrepType`, which itself directly extends `VrmlType`, is an object that describes some solid geometry that can be displayed and explored in a VRML browser. The primitive shape types are defined in *empirical worlds* as `CadnoBox` for a VRML `Box` node, `CadnoCone` for a VRML `Cone` node and so on. In this chapter, new syntax is introduced to describe values on the right-hand side of explicit definitions where no suitable VRML syntax is available. Each object that extends `FrepType` must implement a method `f()` that is the *function representation*[4] for a point set of the geometric shape represented by an instance of the object. This method should return a floating point value for any three float arguments `(x, y, z)` representing a point in space consistent with the condition below.

$$\texttt{f(x, y, z)} \begin{cases} < 0 & \text{if } (x, y, z) \text{ is outside the solid object.} \\ = 0 & \text{if } (x, y, z) \text{ is on the surface of the solid object.} \\ > 0 & \text{if } (x, y, z) \text{ is inside the solid object.} \end{cases}$$

The VRML-2 notation supports the affine transformation of nodes. These are provided by the `Transform` node described in Section 7.3.6. If shapes other

[3]Only a few simple classes are implemented that directly extend `VrmlType` at this time. Future work on *empirical worlds* should extend these to include all VRML-2 nodes.

[4]See Section 3.4.2 and [PASS95].

than the primitive VRML shapes are required by the designer of a world, they must either transform the nodes to appropriate positions and overlap them[5], or calculate a polygonisation of the complex geometric object. Such a polygonisation is a set of edge-connected, filled, three-dimensional triangles oriented in space so that they approximate the surface of the solid object. Sets of triangles can be rendered efficiently by modern computer graphics hardware that is optimised for displaying these triangulations [WND97].

To enrich the range of geometric shapes available in *empirical worlds*, new transformations and operators on point sets are introduced based on the function representation of the objects. This is combined with a generic polygonisation algorithm for all classes that extend `TopologyType`. This abstract class directly extends `FrepType` and implements a `virtualise()` method that calculates a polygonisation from the function representation of its subclasses. Classes that extend `TopologyType` use the function representation binary operators for set-theoretic operations and blends to construct new geometric shapes. The polygonisation algorithm generates a `String` of VRML containing all the triangles that approximate the new geometric shape. All classes that extend `TopologyType` also extend `FrepType` and so they must implement the method `f()` and have their own function representation.

In the following section, controlling the rendering level for the polygonisation algorithm using the `CadnoDetail` data type and the `CadnoMakeDetail` operator is described. As well as being an important introduction to the process of controlling efficient interaction with *empirical world* scripts, this data type and operator is used to illustrate the syntax of the descriptions given for other data types and operators throughout this chapter, Chapter 8 and Appendices A.4 and A.5. Polygonalisation

[5] In most VRML browsers, this produces the effect that the objects appear to be the union of point sets for primitive shapes.

is only used when necessary and if a geometric object is composed of primitives available in VRML then the visualisation output consists of appropriate VRML rather than a list of polygons. Section 7.3 describes classes that represent the primitive geometry available within VRML, including the affine transformations of VRML. In Section 7.4, binary and n-ary operations on point sets for set-theoretic operations, blending and morphing are described.

7.2.1 Rendering Detail

When an object that is a subclass of `TopologyType` is rendered by a polygonisation of its shape, the detail to which that polygonisation is carried out can be controlled by an associated `CadnoDetail` object. The three numbers following `detail` in the string value of the data type correspond to the number of space-dividing voxels that a function representation `f` of a geometric shape is split into for sampling, with edges parallel to the x, y and z axis of the space. The polygonalisation algorithm in the *empirical worlds* softare implements cubic cell polygonalisation in a similar way to that described in [BBCG+97] and further details of the polygonalisation algorithm can be found in Section 8.4.3.

In this thesis, a table with a single line above and below its contents is used to represent *empirical world* classes that extend `JaM.DefnType` (see Section 6.2.2). These tables illustrate the integration of VRML-like syntax into *empirical worlds*. The table for the `CadnoDetail` class is shown below:

Class Name	`CadnoDetail`
Extends	`DefnType`
Value Format	`detail` *integer integer integer*
Default Value	`detail 10 10 10`

In this table, the rows describe the following information:

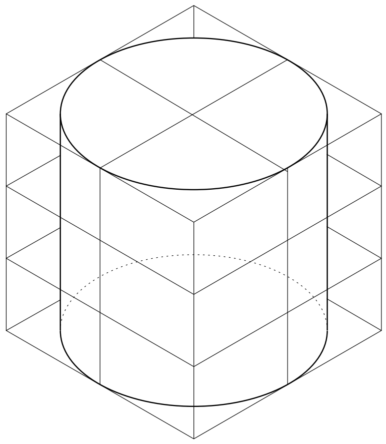


Figure 7.3: Voxels surrounding a cylinder point set.

Class Name The name of the *empirical worlds* class.

Extends The name of the class that the class represented directly extends.

Value Format A representation of the syntax for the description of a value of the data type for the right-hand-side of an explicit definition in an *empirical world* script. Italicised tokens should be replaced by appropriate values in an *empirical world* script.

Default Value If the value of a variable is not explicitly given by an explicit definition, or an abbreviation is available for describing a value, then this is the default value associated with an instance of the data type.

Figure 7.3 shows a cylinder contained in twelve voxels. For this example, the value represented by the instance of the `CadnoDetail` class is "`detail 2 2 3`". The bounding box of the geometry has been split into two in the x direction, two in the y direction and 3 in the z direction. The polygonisation algorithm for this object would sample the function `f` for the cylinder at the intersection and corner points of the voxel describing thin lines in the figure, including those not visible due to the solid material of the cylinder.

An operator called `CadnoMakeDetail` is available in *empirical world* scripts to define the value of a `CadnoDetail` to be dependent on other *integer* values. Operators are represented by *empirical world* classes that extend `JaM.DefnFunc` (see Section 6.2.3 for more details). This operator allows the user to indivisibly link a defining parameter for some geometry to the level of detail at which an object is rendered. The first argument to the operator is the number of voxel divisions for the bounding box parallel to the x-axis, the second is the number of divisions along the y-axis and the third is the number of divisions along the z-axis.

In this thesis, a table with double-lines above and below its contents is used to describe an operator class that extends `JaM.DefnFunc`. These tables represent the way in which dependencies can be expressed between VRML-like values through implicit definitions in *empirical worlds*. The table representing the `CadnoMakeDetail` class is shown below:

Operator Class		`CadnoMakeDetail`
Maps From		*integer* × *integer* × *integer*
Maps To		`CadnoDetail`
Example	Defn.	`d1 = detail(10, 20, 23)`
	Value	`d1 = detail 10 20 23`

In this table, the rows describe the following information:

Operator Class The name of the *empirical world* class that represents an operator in the *empirical world* notation.

Maps From Description of the data types possible in the sequence of arguments.

Maps To Data type associated with the identifier on the left-hand side of an implicit definition that is based on the operator.

Example An example of the operators use, split into:

Defn. an implicit definition containing the operator;

Value an explicit definition that corresponds to the example implicit definition, as it associates the same value with the identifier on the left-hand side.

The implicit definition "`d1 = detail(10, 20, 30)`" defines the value associated with identifier `d1` to be the same as the explicit definition "`d1 = detail 10 20 30`".

7.3 Primitive Shapes in Empirical Worlds

All primitive shapes in *empirical worlds* are data types represented by subclasses of `FrepType`. They have both a VRML string representation of a VRML node and also a function representation corresponding to the equivalent point set. The primitive solid geometric types presented in this section are `Box`, `Cylinder`, `Cone` and `Sphere`, the same as the primitive shape types in VRML.

All primitive shapes in VRML and *empirical worlds* are defined so that their dimensions are centred at the origin[6]. To move a primitive away from the origin, it is necessary to use the affine transformation data type (an instance of `CadnoTransform`) that represents transformed geometry. The `CadnoTransform` data type allows the combination of any rotations, scaling, reflections, translations and linear shears on any list of geometric objects represented by classes that extend `FrepType`.

As the `CadnoTransform` can be exactly described by a string representing a VRML node, it is treated along with the primitive shapes and the `FrepList` in this section as a subclass of `FrepType` for which the `virtualise()` method returns

[6]The origin is not necessarily the centre of gravity for a solid. Consider a `Cone` primitive with more material below the origin than above the origin.

```
Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0.2 0.8 0.6
    }
  }
  geometry Transform {
    rotation 0 0 1  1.57
    children [
      Box { size 1.0 2.0 3.2 },
      Cylinder { radius 3.1 height 4.5 },
      Cone { bottomRadius 2.1 height 3.3 },
      Sphere { radius 9.3 }
    ]
  }
}
```

Table 7.2: VRML-2 file containing *primitive shape* nodes.

a string of VRML that does not contain a polygonisation of a shape. The way in which classes in this section are rendered is determined by the VRML browser in which their shape is viewed. Table 7.2 shows an example VRML-2 file containing all the primitives documented in this section of the chapter, along with an `Appearance` node that is a special attribute node described further in Appendix A.4.

7.3.1 Box Point Sets

A box point set has six rectangular faces, eight vertices and is parametrised by its length (`a0`) along the x-axis, width (`a1`) along the y-axis and height (`a2`) along the z-axis. Each face is parallel to either the $x = 0$, $y = 0$ or $z = 0$ planes. A diagram of a box with parameter "`size a0 a1 a2`" is shown in Figure 7.4. In *empirical worlds*, the centre of the material contained within the box is the origin. Every instance of a subclass of `FrepType` has a field to represent a bounding box. The minimum coordinate of a `Box` in each dimension is $(\frac{-a0}{2}, \frac{-a1}{2}, \frac{-a2}{2})$ and the maximum coordinate is $(\frac{a0}{2}, \frac{a1}{2}, \frac{a2}{2})$. These coordinates define the bounding box for `Box` shapes.

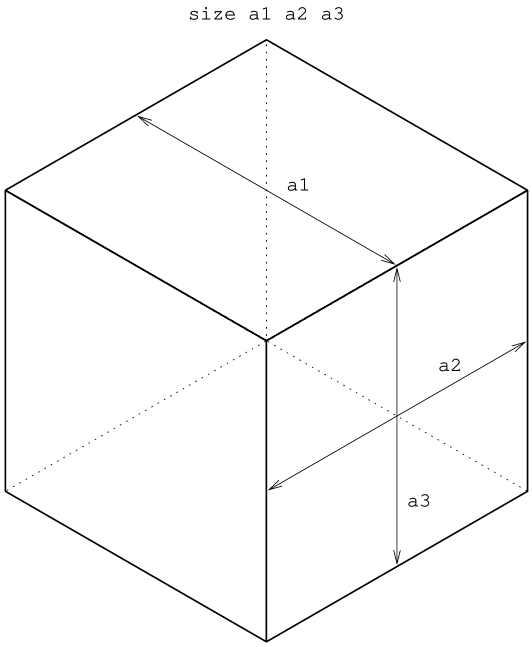


Figure 7.4: Diagrammatic representation of a `Box` point set.

Class Name	`CadnoBox`	
Extends	`FrepType`	
Value Format	`Box` { `size` *point* }	
Default Value	`Box` { `size 2 2 2` }	
Parameters	Name	Type
	`size`	`Point`

In an explicit definition for a box, there can be optional parameters between curly braces "{" and "}". All primitive shape types in *empirical worlds* have a **Default Value** for all defining parameters. The deafult value corresponds to the value of a parameter if it does not explicitly appear between the curly braces in an explicit definition in an *empirical world* script. In the case of the `Box`, the expression of a value "`Box` { }" is equivalent to the expression "`Box` { `size 2 2 2` }"[7]. The data type of the parameters that appear within the curly braces is shown in the **Parameters** section of the table, where the parameter *Name* is next to its *Type* in

[7]This is the same convention as in VRML, where most nodes have a default value.

terms of other *empirical world* classes. The grammar that matches this data type identifies the parameter from its string.

Operator Class		`CadnoMakeBox`
Maps From		*float* × *float* × *float* or *integer* × *integer* × *integer* etc.
Maps To		`CadnoBox`
Example	Defn. Value	`b1 = box(3.2, 4, 5.6)` `b1 = Box { size 3.2 4 5.6 }`

Class `CadnoMakeBox` describes an operator that takes three arguments that represent the length, width and height of a box and returns an instance of a `CadnoBox` class with these dimensions, centred at the origin. This is the mechanism for establishing dependencies between boxes and other geometry. It is impossible to have a box with negative length sides and so the absolute value is taken for any negative parameters for a box during an update.

The function representation of a box point set is defined as the intersection of six *half spaces*. For any point (x, y, z) a possible function representation for a box f is given in the formulae

$$h(u,v) = -\left(u - \frac{v}{2}\right)\left(u + \frac{v}{2}\right) \tag{7.1}$$

$$i(u,v) = u + v - \sqrt{u^2 + v^2} \tag{7.2}$$

$$f(x,y,z) = i(i(h(x,a0), h(y,a1)), h(z,a2)) \tag{7.3}$$

The mapping $h(u, v)$ is the function representation for the intersection of two half spaces, with normal vectors oriented along the u-axis, separated by distance v and equidistant from the plane $u = 0$. The mapping $i(u, v)$ is a function representation for the intersection of two other function representation values at the same point u and v.

The function implemented in the method `f` for the `CadnoBox` object has C^1 discontinuity only at the edges of the box. Smoother geometry and aesthetically pleasing images are created for operators that combine point sets, such as blends, for function representations that are C^1 continuous everywhere except shape edges.

7.3.2 Sphere Point Sets

Sphere point sets are centred on the radius and parametrised by a radius `r`. The bounding box for a sphere is given by the minimum point of $(-r, -r, -r)$ and the maximum point (r, r, r). Figure 7.5 shows a diagrammatic representation of a sphere point set, where the three small arrows at its centre point are oriented along the three axes of the space. (Each arrow has the end point of its tail located at the origin.)

Class Name	`CadnoSphere`	
Extends	`FrepType`	
Value Format	`Sphere` { `radius` *float* }	
Default Value	`Sphere` { `radius 1.0` }	
Parameters	Name	Type
	`radius`	`CadnoFloat`

Class `CadnoMakeSphere` extends `DefnFunc` and allows for the implicit definition of `CadnoSphere` instances in *empirical worlds*.

Operator Class		`CadnoMakeSphere`
Maps From		*float* or *integer*
Maps To		`CadnoSphere`
Example	Defn.	`s1 = sphere(3.7)`
	Value	`s1 = Sphere` { `radius 3.7` }

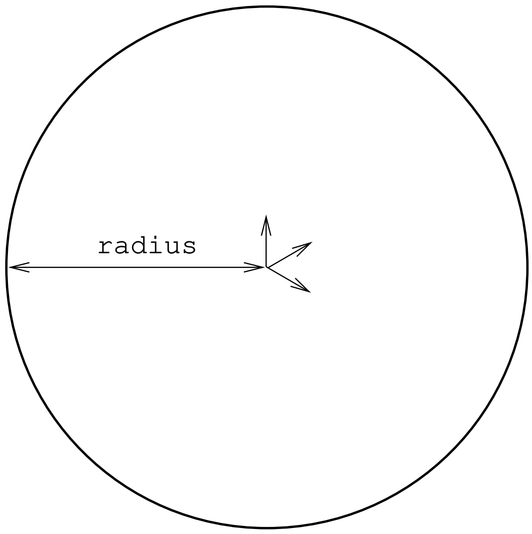


Figure 7.5: Diagrammatic representation of a `Sphere` Point Set

The implemented function representation for a `CadnoSphere` object for any point (x, y, z) in three-dimensional space with radius r is given by the equation.

$$f(x, y, x) = r^2 - (x^2 + y^2 + z^2) \tag{7.4}$$

This function has C^1 continuity everywhere.

7.3.3 Cylinder Point Sets

Cylinder point sets are parametrised by their height (`h`) and radius (`r`). In *empirical worlds* and VRML, the central axis along which the height is measured, is oriented along the y-axis. The cylinder is centred at the origin and has a bounding box with minimum point $(-r, \frac{-h}{2}, -r)$ and maximum point $(r, \frac{h}{2}, r)$. A diagrammatic representation of a cylinder centred at the origin is shown in Figure 7.6.

Class Name	`CadnoCylinder`
Extends	`FrepType`

Value Format	`Cylinder` {
	 `height` *float*
	 `radius` *float*
	 }

Default Value	`Cylinder` {
	 `height 2.0`
	 `radius 1.0`
	 }

Parameters	Name Type
	 `height` `CadnoFloat`
	 `radius` `CadnoFloat`

Two parameters are used in an explicit definition of a cylinder. The order in which they appear is not important when the string representing a value is parsed to create and instance of a `CadnoCylinder`. If one of the parameters is missing then the default value for that one parameter is assumed. For example, "`Cylinder` { `height 4.5` }" is equivalent to "`Cylinder` { `height 4.5 radius 1` }". If an *integer* value is given for one of the parameters, it is converted from the internal representation of the cylinder to a floating point value.

Class `CadnoCylinder` represents an operator in *empirical world* scripts for the implicit definition of cylinder shapes. There are two arguments to this operator — the first represents the height of the cylinder and the second represents the radius.

Operator Class		`CadnoMakeCylinder`
Maps From		*float* $\times$ *float* or *integer* $\times$ *integer* etc.
Maps To		`CadnoCylinder`
Example	Defn. Value	`c1 = cylinder(3.2, 3.4)` `c1 = Cylinder` { `height 3.2 radius 3.4` }

A function representation for a cylinder centred at the origin with central

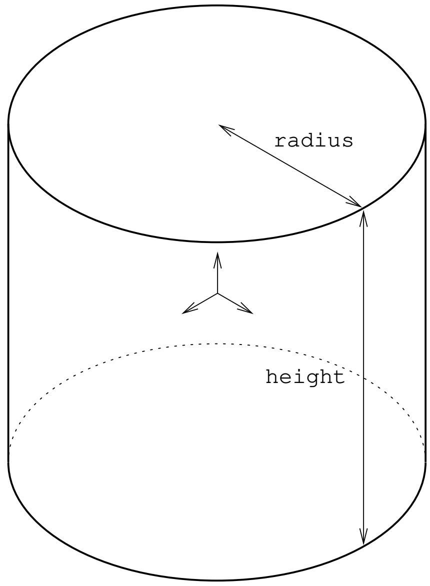


Figure 7.6: Diagrammatic representation of a `Cylinder` point set.

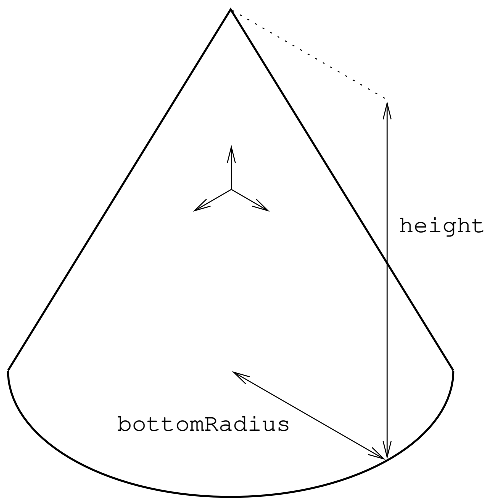
axis set as the y-axis can be defined by the intersection of an infinite cylinder and two half spaces with normal vectors along the y-axis. The function representation f for a cylinder at for any point (x, y, z) is

$$\begin{aligned} f(x,y,z) &= \Big(r^2 - (x^2+z^2)\Big) + \Big(-(y-\frac{h}{2})(y+\frac{h}{2})\Big) \\ &\quad -\sqrt{(r^2-(x^2+z^2))^2 + \Big(-(y-\frac{h}{2})(y+\frac{h}{2})\Big)^2} \end{aligned}$$

This function representation is C^1 continuous except at the edges of its top and bottom faces.

7.3.4 Cone Point Sets

Cone points sets are parametrised by a radius (`r`) for the bottom circle of the cone and a height (`h`) with its central axis oriented along the z-axis. The axis of the cone is centred on the origin and has its top summit point located at $(0, \frac{h}{2}, 0)$. The minimum point of the bounding box for the point set is $(-r, \frac{-h}{2}, -r)$ and the



maximum point is $(r, \frac{h}{2}, r)$. A diagrammatic representation of a cone is shown in Figure 7.3.4, where the `height` and `bottomRadius` parametrisations are indicated.

Class Name	`CadnoCone`	
Extends	`FrepType`	
Value Format	`Cone` { `bottomRadius` *float* `height` *float* }	
Default Value	`Cone` { `bottomRadius 1.0` `height 2.0` }	
Parameters	Name	Type
	`bottomRadius`	`CadnoFloat`
	`height`	`CadnoFloat`

Class `CadnoMakeCone` extends `DefnFunc` and represents an operator that allows an instance of `CadnoCone` to be implicitly defined in an *empirical world* script. The first argument to the operator is the `bottomRadius` of the cone and the second argument is the `height`.

Operator Class		`CadnoMakeCone`
Maps From		*float* $\times$ *float* or *integer* $\times$ *integer* etc.
Maps To		`CadnoCone`
Example	Defn.	`c2 = cone(2.3, 7)`
	Value	`c2 = Cone { bottomRadius 2.3 height 7 }`

The function representation f of a cone with height h and bottom radius r can be defined as the intersection between two half spaces and an infinite cylinder whose radius varies in linear proportion to the y-coordinate of the sampling point. In the formulae for the function representation of a cone below: mapping *rad* is used to calculate the varying radius, mapping *half* returns the half spaces used for the intersection and mapping *cyl* is an infinite cylinder along the z-axis with a varying radius. For any point (x, y, z), the cone f is given by the formulae

$$\begin{aligned}
rad(y) &= \frac{-ry}{h} + \frac{r}{2} \\
cyl(x,z,q) &= q^2 - (x^2 + z^2) \\
half(y) &= -(y - \frac{h}{2})(y + \frac{h}{2}) \\
f(x,y,z) &= half(y) + cyl(x,z,rad(y)) - \sqrt{half(y)^2 + cyl(x,z,rad(y))^2}
\end{aligned}$$

The function representation f for the cone has C^1 discontinuity only at the edge of its base and at its summit point.

7.3.5 Lists of `FrepType` Geometric Objects

In VRML, the `Group` node allows nodes of various kinds to be grouped into one node. An example of a group node is shown in Table 7.3 where a `Box`, `Cylinder` and `Cone` have been grouped into one VRML node by placing them in a comma separated list between "`children` [" and "]". Many other nodes in VRML, such as the `Transform` node in the script in Table 7.2, use lists starting with the token

```
Group {
  children [
    Box { size 2 3 4.2 }, Cylinder { },
    Cone { bottomRadius 3 }
  ]
}
```

Table 7.3: A grouping node in VRML-2.

"`children`" as a parameter for grouping the internal structure of the node. Groups of nodes with interfering points sets are typically drawn one on top of the other in a VRML browser. The resulting virtual-world objects look like a set-theoretic union, although if elements of the group have different appearances or textures specified then this can lead to a visually unsatisfactory model.

In *empirical worlds*, it is not possible to place any node in a list in the way that it is in a `Group` node in VRML. A distinction is made between data type classes that extend `FrepType` and other classes that extend `DefnType` without extending `FrepType`. This is so that it is clear which data types correspond to graphically representable solid geometry and which data types are related to texture, appearance, sound, viewpoint and so on. The `FrepList` is a special kind of list that can contain only nodes (*empirical world* notation types) that extend `FrepType`[8]. This also allows operators that map from shapes described by function representations to map from a list[9].

Class Name	`FrepList`
Extends	`FrepType`
Value Format	`children` [$(FrepType)^*$]
Parameters	A list of objects that extend `FrepType`.

[8]Lists in *empirical worlds* are not comma-separated as they are in VRML.

[9]This limitation of *empirical worlds*, which is implementation specific, should be a future topic for research and improvement.

This class illustrates an important feature of the JaM Machine API. A list of geometric shapes has an explicit value and is its own data type, for example "`children [ Box { } Cylinder { } ]`". Whether the elements of the list explicitly defined or implicitly defined by the operator represented by class `CadnoMakeList`, the value associated with the left-hand side identifier can be represented by a unique string. Further operators for lists (`CadnoHead`, `CadnoTail` and `CadnoElementAt`), are described in Appendix A.4.

Operator Class		`CadnoMakeList`
Maps From		(`FrepType`)*
Maps To		`FrepList`
Example	Defn.	`l1 = list(Box { size 1 2 3 }, Cylinder { })`

	Value	`l1 = children [`
		`Box { size 1.0 2.0 3.0 }`
		`Cylinder { height 2.0 radius 1.0 }`
		`]`

The function representation of an `FrepList` f is the maximum value of the function representation for all the elements of the list $g_1, \ldots, g_n$ at any point (x, y, z), as described by the equation

$$f(x, y, z) = \max(g_1, \ldots, g_n) \tag{7.5}$$

The function representation f represents the set-theoretic union of all the point sets represented by the function representations in the list. Note that the intended use of lists is to group shapes together and not to construct set-theoretic unions. This shape representation of this class can produce similar results to the geometric object overlap that occurs in VRML groups, with the function f having C^1 continuity everywhere except locations where the component function representations have equal values[10]

[10]Detail inside a solid shape is preserved by VRML groups and lost in *empirical world* lists.

7.3.6 Affine Transformations of Point Sets

All the primitive shape points sets in VRML and *empirical worlds* are constructed centred at the origin. A virtual world full of primitive shape point sets centred at the origin would not be a particularly interesting one. In the real world, objects are centred in many different locations. An affine transformation of any geometric solid object allows for its point set to be relocated anywhere in virtual space, rotated around any axis and scaled in a particular orientation to an appropriate size. VRML has a `Transform` node that can be used for the combined purpose of rotation, translation and scaling for any piece of geometry[11]. The analogue to this node in *empirical world* classes is an instance of `CadnoTransform`.

This section explains the process of integrating the VRML `Transform` node syntax with the function representation of shape in a definitive script. This explanation is presented in the form of an annotated transformation of a two-dimensional hexagonal polygon.

Figure 7.7 shows the effect of a set of affine transformations on a hexagon in two dimensions. The transformations shown are carried out in the same order as the transformations in both VRML and instances of the *empirical world* `CadnoTransform` class. Vector $\mathbf{c}$ is the central point for the transformations, vector $\mathbf{t}$ is the translation vector of the geometric objects, $\mathbf{O}$ is a matrix representing a rotation prior to a scaling, $\mathbf{S}$ is a diagonal matrix for the actual scaling with $\mathbf{c}$ at its centre and $\mathbf{R}$ is a rotation matrix representing the required rotation of the object about centre $\mathbf{c}$[12].

In the diagram in Figure 7.7, each stage of the transformation is represented by separate pairs of axes that are labelled "`a`" through to "`g`". On each pair of axes, the hexagonal shape is drawn with a solid line to show it prior to transformation, and

[11] See Table 7.2 for an example of a VRML `Transform` node.

[12] Rotation in two dimensions is about a point, not an axis as in three dimensions. The direction of the axis of rotation in three dimensions is contained in the internal representation of $\mathbf{R}$.

with a dashed line to illustrate the effect of the transformation. Each transformation is described by items in the list below, where each item is labelled to correspond with Figure 7.7:

- a - The point set (hexagon) is translated by vector **-c** so that the other transformations centred at the origin can take place as if they were centred at **c**. The rotated and scaled point set is transformed back by vector **c** in step f.
- b - Rotation of the point set by the inverse scale orientation matrix $\mathbf{O}^{-1}$ prior to scaling. In the example, $\mathbf{O}^{-1}$ corresponds to an anti-clockwise rotation around the origin by $\frac{\pi}{2}$ radians.
- c - Linear scaling of the point set by a factor of 1.5 along the x-axis and 2 along the y-axis. This scale transformation can be represented by a diagonal matrix **S**.
- d - Rotation of the point set by the scale orientation matrix **O**.
- e - Rotation of the point set by the rotation matrix **R**. In this example, **R** corresponds to a clockwise rotation about the origin by $\frac{\pi}{3}$ radians.
- f - Translation of the point set from the origin back to the centre of the transformations at point **c**.
- g - Translation of the point set by the given translation vector **t**.
- h - The point set in its new transformed location after all the combined affine transformations are completed.

A bounding box for a translated shape is found by translating the bounding box for the transformed geometric objects and then creating a new box with faces with normal vectors parallel to the axes. This new bounding box completely

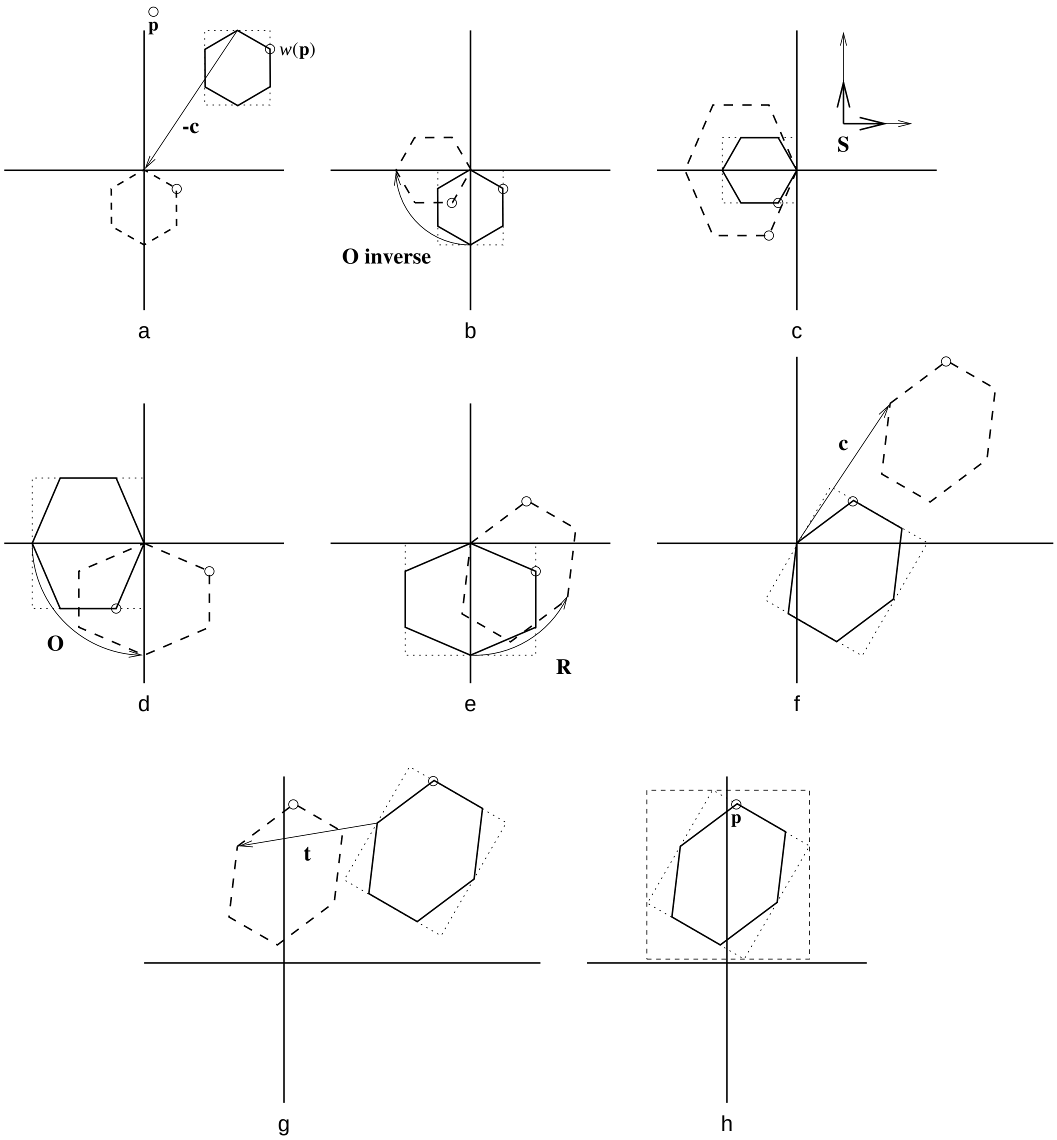


Figure 7.7: Stages for an affine transformation of a hexagonal shape.

contains the translated bounding box. In Figure 7.7, this process is illustrated in two dimensions. In Figure 7.7a, the dotted box represents the original bounding box for the hexagon. The translation of this bounding box is shown in Figures 7.7b through to 7.7g. In Figure 7.7h the new bounding box for the translated geometry is calculated, with sides that have normal vectors along the x and y axis. This new bounding box is represented by a box drawn with a dashed outline.

Bounding boxes constructed by this process are often larger than the minimum box that can tightly enclose the geometry. Further work is required to identify better mechanisms for determining the minimum bounding box for a transformed shape in *empirical worlds.* The bounds are primarily an aid to rendering algorithms, providing a guide to the spatial extent of a solid shape given by a function representation.

Class Name	`CadnoTransform`
Extends	`FrepType`

Value Format	`Transform` {
	 `center` *point*
	 `rotation` *rotation*
	 `scale` *point*
	 `scaleOrientation` *rotation*
	 `translation` *point*
	 `children` [(*FrepType*)*]
	 }

Default Value	`Transform` {
	 `center 0 0 0`
	 `rotation 0 0 1 0`
	 `scale 1 1 1`
	 `scaleOrientation 0 0 1 0`
	 `translation 0 0 0`
	 `children` [ ]
	 }

Parameters	Name — Type
	 `center` — `Point`
	 `rotation` — `CadnoRotation`
	 `scale` — `Point`
	 `scaleOrientation` — `CadnoRotation`
	 `translation` — `Point`
	 `children` — List of objects of `FrepType`.

The `CadnoTransform` class represents three-dimensional versions of affine transformations, with **c** represented in the `center` parameter, **R** in the `rotation` parameter, **O** in this `scaleOrientation` parameter, **S** in the `Scale` parameter and **t** in the `translation` parameter. Once a string represented by a `CadnoTransform` has been parsed, the internal data representation of these parameters are computed as matrices and vectors. This minimises the number of computationally expensive calls to mathematical library functions for sine, cosine and square root during the evaluation of the functional representation of a transformed point set. Note that the default value for a `CadnoTransform` corresponds to the identity mapping for each component transformation, e.g. a `translation` vector of $(0,0,0)$, a rotation through 0 degrees and so on.

The function representation for a transformed point set is found by transforming the original space for the original function representation of the point set. To evaluate the function representation of the transformed shape, any point in the space of the transformed shape is mapped by an affine transformation to the original space, in which the function representation of the original shape is evaluated. This transformation of space is represented by the function w, which corresponds to the inverse of the shape transformation represented, where

$$w(\mathbf{p}) = (\mathbf{O}\mathbf{S}^{-1}\mathbf{O}^{-1}\mathbf{R}^{-1}((\mathbf{p} - \mathbf{t}) - \mathbf{c})) + \mathbf{c} \tag{7.6}$$

For any point $\mathbf{p} = (x, y, z)$ and a function representation for the `children` of a `CadnoTransform` instance g, the function representation for the transformed point set f is $f(\mathbf{p}) = g(w(\mathbf{p}))$.

In Figure 7.7, one of the vertices of the hexagon shape is circled in each representation to illustrate this space transformation process. Point $\mathbf{p}$ is shown in transformed space in Figure 7.7h, and is located at a vertex of the transformed hexagon. To evaluate the function representation of the transformed hexagon at $\mathbf{p}$, $\mathbf{p}$ is transformed by w back through Figures 7.7g to 7.7a where the function representation for the original shape is evaluated at $w(\mathbf{p})$.

Four operators can be used in *empirical world* scripts to implicitly define instances of the `CadnoTransform` class. These are: one for translating point sets, one for rotating point sets, one for scaling/linearly shearing point sets and one for a combination of all these transformations.

Implicit Definitions for Translations

Operator Class		`CadnoTformTranslation`
Maps From		*float* × *float* × *float* × *FrepType* or *integer* × *integer* × *integer* × *FrepType* etc.
Maps To		`CadnoTransform`
Example	Defn.	`t1 = translate(2.3, 2.4, 2.5,` `        children [ Box { size 2 3 1 } ] )`

	Value	`t1 = Transform {`
		`        center 0 0 0`
		`        rotation 0 0 1  0`
		`        scale 1 1 1`
		`        scaleOrientation 0 0 1  0`
		`        translation 2.3 2.4 2.5`
		`        children [ Box { size 2 3 1 } ]`
		`        }`

The translation of a point set can be implicitly defined in *empirical worlds* using the operator represented by the `CadnoTformTranslation` class. The data type of the value associated with the left-hand side of an implicit definition containing this operator is represented by the `CadnoTransform` class. The first three arguments to the operator represent the translation vector **t** and the fourth argument is the original point set prior to translation.

Implicit Definitions for Rotations

Operator Class		`CadnoTformRotation`
Maps From		`Point` × `CadnoRotation` × *FrepType* or `CadnoRotation` × *FrepType*
Maps To		`CadnoTransform`
Example	Defn.	`t2 = rotate(2 2 3, 0 1 0  3.1415,` `      children [ Cone { } ] )`

	Value	`t2 = Transform {`
		`      center 2 2 3`
		`      rotation 0 1 0   3.1415`
		`      scale 1 1 1`
		`      scaleOrientation 0 0 1   0`
		`      translation 0 0 0`
		`      children [`
		`        Cone { bottomRadius 1 height 2} ]`
		`      }`

Example	Defn.	`t3 = rotate(0.5 0.25 0.25   1.5708,` `      Sphere { radius 3.2 })`

	Value	`t3 = Transform {`
		`      center 0 0 0`
		`      rotation 0.5 0.25 0.25   1.5708`
		`      ⋮`
		`      children [ Sphere { radius 3.2 } ]`
		`      }`

The class `CadnoTformRotation`, when used as an operator for an implicit definition in an *empirical world* script, creates instances of `CadnoTransform` class representing the given geometric shapes rotated by the specified amount. The first argument to the operator is an optional vector represented as a `Point` that is the centre ($\mathbf{c}$) of the transformation and, therefore, a point that the axis of rotation must pass through. The next argument is a `CadnoRotation` representation ($\mathbf{R}$) (see Appendix A.4) of the axis for the rotation and the anti-clockwise angle of rotation. The last argument is the function representation for the point set for the geometry that is being transformed.

In the table describing the `CadnoTformRotation` class above, the second example shows the rotation of a `CadnoSphere`. The resulting point set is actually identical to the original one. With all objects in *empirical worlds*, many different descriptions can exist for the same point set. The difference between the representations of the point sets can be established by comparing their string representations. Any user interacting with a script of *empirical world* definitions should take care to ensure they consider the most efficient way to represent their point sets by examining the issues relating to dependency structures discussed in Chapter 4.

Implicit Definitions for Scaling

Operator Class		`CadnoTformScale`
Maps From		`Point` × *FrepType* or *float* × *float* × *float* × *FrepType* or `Point` × `CadnoRotation` × `Point` × *FrepType* or `CadnoRotation` × `Point` × *FrepType*
Maps To		`CadnoTransform`
Example	Defn.	`s4 = scale(1, 2, 1,` `        children [ Box { size 2 2 2 } ] )`

	Value	`s4 = Transform {`
		`        ⋮`
		`        scale 1.0 2.0 1.0`
		`        ⋮`
		`        children [ Box { size 2 2 2 } ]`
		`        }`

A scaling transformation of a point set can be described by an implicit definition with the operator represented by class `CadnoTformScale`. There are four possible sequences of types in arguments to this operator and, in each case, the data types associated with the identifier on the left-hand side of the implicit definition is represented by a `CadnoTransform` class. One of these sequences of arguments consists of a `Point` representing the scale transformation **S**, followed by a

`CadnoRotation` representation of scale orientation matrix **O**, followed by a `Point` representing the centre vector **c**. The final argument in the sequence corresponds to the function representation for the shape being transformed.

Implicit Definitions for Affine Transformation

Operator Class		`CadnoMakeTransform`
Maps From		`Point` × `CadnoRotation` × `Point` × `CadnoRotation` × `Point` × *FrepType*
Maps To		`CadnoTransform`
Example	Defn.	`t5 = transform(2 2 3, 0 1 0  3.1415, 1 2 1, 1 0 0  1.5708, 2 3 4, children [ Box { size 2 4 2 } ] )`
	Value	`t5 = Transform { center 2 2 3 rotation 0 1 0  3.1415 scale 1 2 1 scaleOrientation 1 0 0  1.5708 translation 2 3 4 children [ Box { size 2 4 2 } ] }`

Every parameter for an instance of a `CadnoTransform` class can be implicitly defined in an *empirical world* script by using the operator represented by the `CadnoMakeTransform` class. The ordering of the sequence of arguments to the operator is: `center` **c**, `rotation` **R**, `scale` **S**, `scaleOrientation` **O** and `translation` **t**. The last argument of this sequence corresponds to the function representation for the original point set that is being transformed.

7.4 Shape Combination in Empirical Worlds

In this section, binary and n-ary operations for point sets are described. These form new data types and operators in *empirical world* scripts. They include types for the description of, and operators for the implicit creation of, new shapes that are a combinations of other existing point sets. The new point sets can also be used as parameters to further combination operations. The combinations described here are the three standard Constructive Solid Geometry[13] operators [Bow94] of union, intersection and difference (also known as cut) of point sets, along with two variations of these, blend-union and blend-intersection. Finally, an operation representing the metamorphosis between two existing point sets is described.

None of the operators described in this section are part of the existing VRML notation and new syntax has been adopted to describe them. The shapes described by the combination operators must be rendered through the polygonisation algorithm described in Section 8.4.3[14]. To distinguish these shapes from those that may have a VRML representation, all subclasses of `DefnType` described here also extend another abstract class called `TopologyType`. The relationship between these classes is shown in the class diagram in Figure 7.2. All objects that extend `TopologyType` have the same `virtualise()` method to render their geometry and must implement a method `f` corresponding to their function representation for any three-dimensional point (x, y, z).

The classes that represent operators described in Section 7.4.1 through to the end of this chapter can take an additional `CadnoDetail` parameter as an argument in *empirical world* scripts (see Section 7.2.1). This parameter allows for the interactive control of the quality of the polygonisation and hence the level of detail of the image displayed in the VRML browser. For all implicit definitions of shape in these sections, the sequence of arguments to the operators can be preceded by an optional argument that is a variable represented by the `CadnoDetail` class.

[13] Commonly abbreviated to CSG, also known as set-theoretic operations.

[14] At the current time, polygonisation is the method adopted for rendering function representation shapes in *empirical worlds*. The same function representation can be used to create a ray-traced image [FvF+93].

7.4.1 Union of Point Sets

In this section, the integration of the set-theoretic union of point sets into *empirical worlds* is described. The only parameters that describe a union in *empirical worlds* are a list of `children` classes that extend `FrepType`. The bounding box of a union is a box that bounds the union of all the bounding boxes of the defining shapes. If the bounding boxes of the defining shapes tightly encloses their geometry then the bounding box of the union point set will also tightly enclose its geometry. Figure 7.8 shows a diagrammatic representation of the union of a `Box` and a `Cylinder` point set.

Class Name	`CadnoUnion`	
Extends	`TopologyType`	
Value Format	`Union { children [ (`*FrepType*`)* ] }`	
Parameters	Name	Type
	`children`	List of objects of `FrepType`.

An explicit definition for a union point set in an *empirical world* script is of the form

```
u1 = Union { children [ Box { } Cylinder { height 3 radius 0.5 } ] }.
```

A definition for a union of point sets has an explicit right-hand side value that directly represents a point set for an instance of a `CadnoUnion` class. Implicit definition of a union point set in an *empirical world* script is possible by using an operator based on the `CadnoMakeUnion` class. This operator creates and maintains dependencies based on its arguments for an instance of a `CadnoUnion` class. The two possible sequences of arguments to this operator are either a sequence of variables represented by the `FrepType` class or a variable associated with an existing instance of the `FrepList` class.

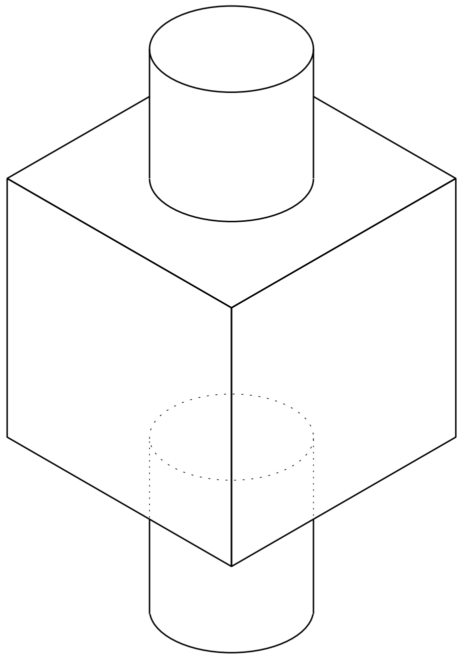


Figure 7.8: Diagrammatic representation of the union of a `Box` and a `Cylinder`.

Operator Class		`CadnoMakeUnion`
Maps From		`FrepList` or `CadnoDetail` × `FrepList` or (*FrepType*)* or `CadnoDetail` × (*FrepType*)*
Maps To		`CadnoUnion`

Example	Defn.	`u1 = union(detail 14 14 14,`
		`children [`
		`Box { size 2 2 2 }`
		`Cylinder { radius 0.5 height 3 }`
		`] )`

	Value	`u1 = Union {`
		`children [`
		`Box { size 2 2 2 }`
		`Cylinder { radius 0.5 height 3 }`
		`] }`

In the table above describing the `CadnoMakeUnion` class, the example shows a union of a `Box` and a `Cylinder` equivalent to that shown in Figure 7.8. If rendered in a VRML browser, the polygonisation used to generate the image of this example shape would split the shape into $14 \times 14 \times 14$ equal voxels that are contained within the bounding box of the union point set.

One possible function representation for a union of point sets with function representations $g_1, \ldots, g_n$ at any point $\mathbf{p} = (x, y, z)$ in space, is to find the maximum value of $g_1(\mathbf{p}), \ldots, g_n(\mathbf{p})$. This representation can have C^1 discontinuity wherever any pair of the arguments are equal. Better continuity can be achieved with an instance of the `CadnoBlendUnion` described in Section 7.4.4 of this chapter, with a value for `displacement` set to zero. The blend union is a binary operation whereas the standard union described in this section is an n-ary operation.

7.4.2 Intersection of Point Sets

In this section, the integration of set-theoretic intersection into *empirical world* scripts is described. The parameters for an intersection of point sets in *empirical worlds* are a list of `children` containing the defining point sets of the intersection. This operation can result in an empty point set if there is no common solid material represented by any of the defining points sets.

The bounding box of an intersection point set is a box that tightly bounds the intersection of all the bounding boxes of the defining point sets. If the bounding boxes for the children tightly enclose their geometry then the bounding box for the intersection geometry will also tightly enclose the geometry. Figure 7.9 shows the intersection of a `Sphere` with a `Cone` rendered using the *empirical world* builder tool (see Section 8.4). The resulting geometry from this operation can be described as a truncated cone with a rounded base and top.

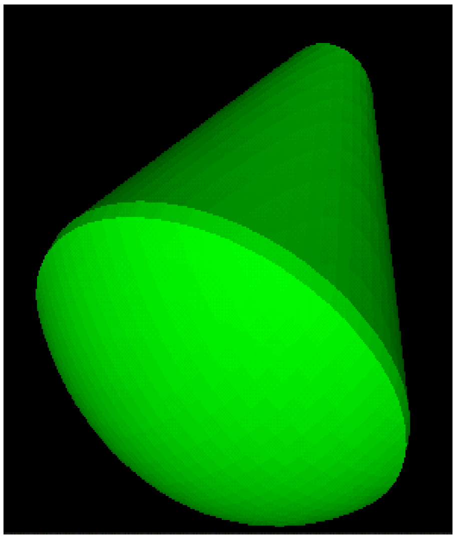
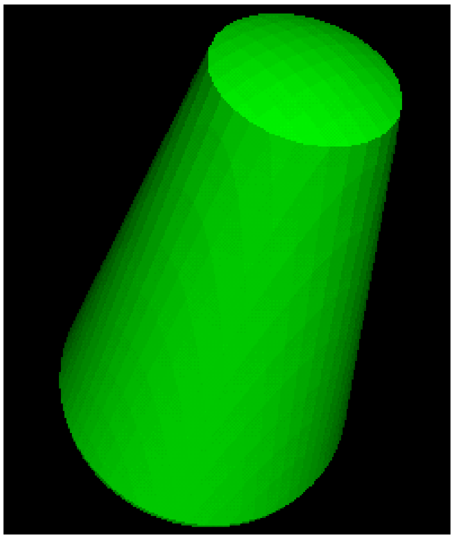
 

Figure 7.9: Images of the intersection of `Cone` and `Sphere` point sets.

Class Name	`CadnoIntersection`	
Extends	`TopologyType`	
Value Format	`Intersection` { [`children` (*FrepType*)*] }	
Parameters	Name	Type
	`children`	List of objects of `FrepType`.

In an *empirical world* script, an intersection can be defined implicitly by using an operator represented by the `CadnoMakeIntersection` class. The two possible sequences of arguments to this operator are either a sequence of variables represented by the `FrepType` class or a variable associated with an existing instance of the `FrepList` class.

Operator Class		`CadnoMakeIntersection`
Maps From		`FrepList` or `CadnoDetail` $\times$ `FrepList` or $(\mathit{FrepType})^*$ or `CadnoDetail` $\times$ $(\mathit{FrepType})^*$
Maps To		`CadnoIntersection`

Example	Defn.	`i1 = intersection(children [`
		`Cone { bottomRadius 1 height 4 }`
		`Sphere { radius 1 }`
		`] )`

	Value	`i1 = Intersection {`
		`children [`
		`Cone { bottomRadius 1 height 4 }`
		`Sphere { radius 1 }`
		`] }`

The *empirical world* script for the images shown in Figure 7.9 is the same as the one in the table above.

If the defining point sets have a functional representations $g_1, \ldots, g_n$ then one possible function to represent the intersection of these point sets for any point $\mathbf{p}$ is to find the minimum value of $g_1(\mathbf{p}), \ldots, g_n(\mathbf{p})$. This function is C^1 continuous except at any point where any pair of the component function representations have equal values. For better continuity, an intersection of point sets represented by the `CadnoBlendIntersection` class described in Section 7.4.5 can by used, with the `displacement` parameter set to zero.

7.4.3 Cutting Point Sets by Other Point Sets

In this section, the integration of set-difference operation into *empirical worlds* is described. A point set representing some solid material can appear to have the material from another point set cut away by the set-difference of the two point sets. The set-difference operation described here is a binary operation, where the original point set is considered the `body` and the cutting point set as the `tool`. This operation can lead to an empty point set in the case where all the material in the body is also contained in the tool. For a body point set B and a tool T, the set-difference operation $B \setminus T$ is the same as $B \cap \neg T$, the intersection of B with every point not inside T.

A bounding box that is guaranteed to enclose a cut point set is the same as the bounding box for the body shape. A tightly enclosing bounding box for the shape may be smaller than this, as material can be removed from the body by the cut operation. The method `shrinkWrap()` implemented in the class `TopologyType` samples the function representation at the faces of the bounding box for the current level of detail. A good guess to the bounding box is initially found before it is grown or shrunk in the x, y and z directions until it approximates a tight bounding box that *wraps* the geometry given by the function representation. This process can go wrong, especially for infinite solid shapes. To prevent continuous looping and searching for shape, the process is limited to a maximum number of iterations. In most cases, the maximum number of iterations is not reached. When the maximum number of iterations is exceeded, this process can lead to the rendered geometric shapes having holes in their surfaces.

Figure 7.10 is a diagrammatic representation of a `Box` point set body cut by a `Cylinder` point set tool. The parameters `body` and `tool` in an instance of a `CadnoCut` class are the geometric solid shapes used in the set difference.

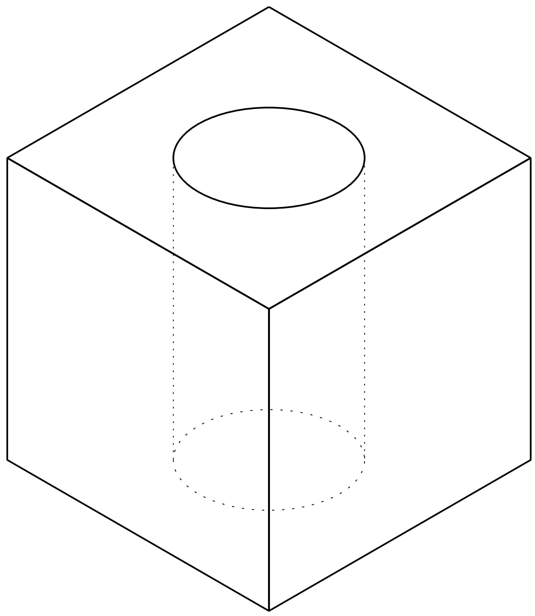


Figure 7.10: Body point set `Box` with a tool point set `Cylinder` cut away from it.

Class Name	`CadnoCut`	
Extends	`TopologyType`	

Value Format	`Cut` {
	 `body` *FrepType*
	 `tool` *FrepType*
	 }	

Parameters	Name	Type
	`body`	`FrepType`
	`tool`	`FrepType`

An instance of a `CadnoCut` class can be implicitly defined in an *empirical world* script by the use of an operator represented by the `CadnoMakeCut` class. The order of arguments to this operator are an optional `CadnoDetail` followed by a solid geometric object representing the `body` for the cut operation and then another solid geometric object that is the `tool`.

Operator Class		`CadnoMakeCut`
Maps From		*FrepType* × *FrepType* or `CadnoDetail` × *FrepType* × *FrepType*
Maps To		`CadnoCut`
Example	Defn.	`c1 = cut(Box { size 2 2 2 },` `        Cylinder { radius 0.5 height 4 } )`

	Value	`c1 = Cut {`
		`        body Box { size 2 2 2 }`
		`        tool Cylinder {`
		`          radius 0.5 height 4 }`
		`        }`

The *empirical world* script example in the table above is consistent with the diagrammatic representation of a `CadnoCut` in Figure 7.10.

The function representation f of one point set cut by another can be found by taking the intersection of the body function representation g_1 with the negative version of the tool function representation g_2 at the same point, where

$$f(x, y, z) = g_1(x, y, z) + (-g_2(x, y, z)) - \sqrt{g_1(x, y, z)^2 + g_2(x, y, z)^2} \qquad (7.7)$$

This function representation has C^1 discontinuity only at a point where the value of g_1 and g_2 are both equal to zero.

7.4.4 Union Blending of Point Sets

Using the function representation of shape, it is possible to describe variants of the set-theoretic operations that create shapes that appear as if they are blended together. Blending with functional representations is described by Savchenko and Pasko in [SP94]. The union of points sets with blending is described in this section and the blending intersection of point sets in Section 7.4.5.

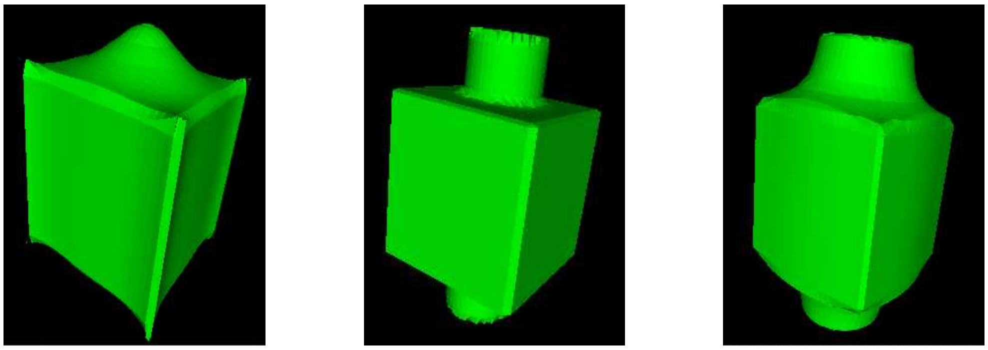
Point set blending involves the addition or subtraction of some material around the outside of the union of the two point sets. To do this with function representations of two geometric shapes g_1 and g_2, and their set-theoretic union u_1, it is necessary to add some blend material in the proximity of both of the two shapes. This can be achieved by adding a little extra of the values of $g_1(\mathbf{p})$ and $g_2(\mathbf{p})$ to the function representation of the value of the union. To do this, the proportion that the blend is affected by each of the defining shapes, a_1 for shape g_1 and a_2 for shape g_2, must be determined. It is also necessary to define the displacement value d that represents how much the blend should affect the union operation as a whole. Then function representation f for a point set resulting from a blend-union operation is given by the formulae

$$u(x,y,z) = g_1(x,y,z) + g_2(x,y,z) + \sqrt{g_1(x,y,z)^2 + g_2(x,y,z)^2} \tag{7.8}$$

$$f(x,y,z) = \frac{d}{1 + \left(\frac{g_1(x,y,z)}{a_1}\right)^2 + \left(\frac{g_2(x,y,z)}{a_2}\right)^2} + u(x,y,z) \tag{7.9}$$

The blending process produces much better results when the functions g_1 and g_2 are C^1 continuous almost everywhere. Figure 7.11 shows three snapshots of the blend union of a `Box` point set and a `Cylinder` point set. The displacement value d is set to -0.5 in the left-hand image, 0.0 in the central image and 0.5 in the right-hand image. In each case, the values for a_1 and a_2 are set to 1.0. With $d = 0$, the point set is exactly the same as for the set-theoretic union operation.

A blend union is parametrised by a float representing d, the `displacement`, and two floats representing the proportion of blending a_0 and a_1 for each defining point set, called `firstDistance` and `secondDistance` respectively. The bounding box for a blend union is found by starting with the bounding box for the set theoretic union of the defining shapes and executing the `shrinkWrap()` method, which approximates bounds that are just outside the actual geometry.



-0.5 0.0 0.5

Figure 7.11: Blend union of a `Box` and a `Cylinder` for `displacement` values of -0.5, 0.0 and 0.5.

Class Name	`CadnoBlendUnion`	
Extends	`TopologyType`	

Value Format	`BlendUnion` {
	    `firstSolid` *FrepType*
	    `secondSolid` *FrepType*
	    `displacement` *float*
	    `firstDistance` *float*
	    `secondDistance` *float*
	}	

Parameters	Name	Type
	`firstSolid`	`FrepType`
	`secondSolid`	`FrepType`
	`displacement`	`CadnoFloat`
	`firstDistance`	`CadnoFloat`
	`secondDistance`	`CadnoFloat`

Implicit definitions of blend unions in *empirical world* scripts can be made by the use of the class `CadnoMakeBlendU`, which extends `DefnFunc`. The order of arguments to the operator in a script are a floating point value for the `displacement` d, followed by the `firstDistance` floating point value a_1, then the `secondDistance` floating point value a_2, then the `firstSolid` with function representation g_1 and finally the `secondSolid` with function representation g_2.

Operator Class		`CadnoMakeBlendU`	
Maps From		`CadnoDetail` × *float* × *float* × *float* × *FrepType* × *FrepType* or *float* × *float* × *float* × *FrepType* × *FrepType*	
Maps To		`CadnoBlendUnion`	
Example	Defn.	`bu = blendUnion(detail 10 10 10, 0.5, 1.0, 1.0, Box { size 2 2 2 }, Cylinder { radius 0.5 height 4 } )`	
	Value	`bu = BlendUnion { firstSolid Box { size 2 2 2 } secondSolid Cylinder { radius 0.5 height 4 } displacement 0.5 firstDistance 1.0 secondDistance 1.0 }`	

7.4.5 Intersection Blending of Point Sets

The intersection blending of point sets creates a point set similar to a set-theoretic intersection of the same two point sets, except that some material is added or removed to achieve the effect of a blend between the intersecting shapes. The amount of material added is controlled by a displacement parameter d and the proportion of material from each geometric shape defining the blend by parameters a_1 and a_2. For any point $(x,\ y,\ z)$ and function representations for two solid geometric shapes g_1 and g_2, the function f for a blend intersection is

$$u(x,y,z) = g_1(x,y,z) + g_2(x,y,z) - \sqrt{g_1(x,y,z)^2 + g_2(x,y,z)^2} \tag{7.10}$$

$$f(x,y,x) = \frac{d}{1 + \left(\frac{g_1(x,y,z)}{a_1}\right)^2 + \left(\frac{g_2(x,y,z)}{a_2}\right)^2} + u(x,y,z) \tag{7.11}$$

In em empirical world scripts, the explicit value of a blend intersection is given by explicitly specifying two shapes `firstSolid` and `secondSolid`, along with the `displacement` parameter d. The proportion of the effect of each shape a_1 and a_2 on the operation can be expressed in `firstDistance` and `secondDistance` parameters respectively. The bounding box for the shape is found by taking the bounding box for the set-theoretic intersection of the two geometric shapes and then executing the `shrinkWrap()` method to find a more accurate set of bounds to suit the blended shape.

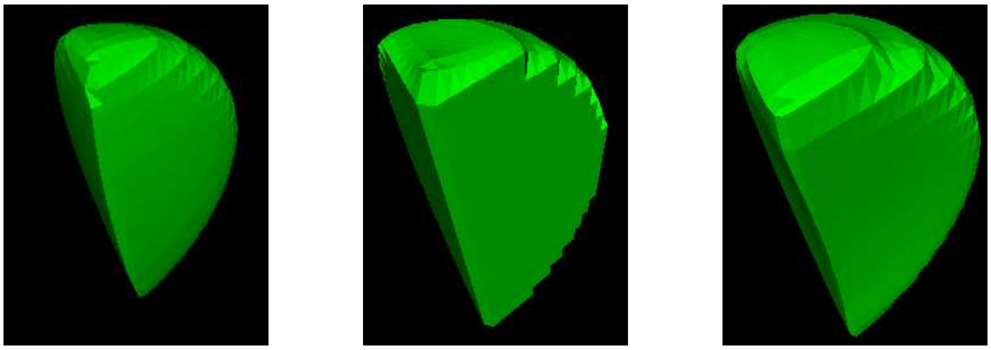
Class Name	`CadnoBlendIntersection`	
Extends	`TopologyType`	

Value Format	`BlendIntersection` {
	 `firstSolid` *FrepType*
	 `secondSolid` *FrepType*
	 `displacement` *float*
	 `firstDistance` *float*
	 `secondDistance` *float*
	 }	

Parameters	Name	Type
	`firstSolid`	`FrepType`
	`secondSolid`	`FrepType`
	`displacement`	`CadnoFloat`
	`firstDistance`	`CadnoFloat`
	`secondDistance`	`CadnoFloat`

Figure 7.12 shows the blend intersection of a sphere and a translated box for different values of the displacement parameter d and with the values of a_1 and a_2 set to 1.0. With a value for $d = 0$, the image is the same as the set-theoretic intersection of the same two shapes. Notice how blend material can be subtracted from the set-theoretic intersection with a negative value for d, as well as added to the intersection with a positive value for d.

Blend intersections in an *empirical world* script can be implicitly defined by the use of the class `CadnoMakeBlendI`, which extends `DefnFunc`. The order of arguments to the operator in a script is the same as the order for the blend union



`-0.1` `0.0` `0.1`

Figure 7.12: Blend intersection of a `Cone` and a `Sphere` for `displacement` values of `-0.5`, `0.0` and `0.5`.

operator.

Operator Class		`CadnoMakeBlendI`
Maps From		`CadnoDetail` × *float* × *float* × *float* × `FrepType` × `FrepType` or *float* × *float* × *float* × `FrepType` × `FrepType`
Maps To		`CadnoBlendIntersection`

Example	Defn.	`ci = blendIntersection(0.5, 1.3, 1.8,`
		`Cone {`
		`bottomRadius 1 height 4 },`
		`Sphere { radius 1 } )`

	Value	`ci = BlendIntersection {`
		`firstSolid Cone {`
		`bottomRadius 1 height 4 }`
		`secondSolid Sphere { radius 1 }`
		`displacement 0.5`
		`firstDistance 1.3`
		`secondDistance 1.8`
		`}`

7.4.6 Metamorphosis of Point Sets

A metamorphosis between point sets, known as a *morph*, is defined as a continuous transition between two pieces of solid geometry parametrised by a value t. When $t =$

0, the morph shape is exactly equal the same as first piece of geometry, and when $t =$ 1 the morph shape is the same as the second piece of geometry. When $t = 0.5$, there is an equal contribution from both pieces of geometry. Such a morphing transition can be achieved by with the function representation for solid objects by the simple equation shown below (Equation 7.12), where g_1 is the function representation for the first solid and g_2 is the function representation for the second. An example of function representation morphing between a box and a spiral by Pasko et al is found in [PSA93]. The morph f is C^1 continuous everywhere that the defining function representations are C^1 continuous, where

$$f(x,y,z) = (1-t)g_1(x,y,z) + tg_2(x,y,z) \tag{7.12}$$

The morph between two solid objects can be explicitly defined in an *empirical world* script through the data type represented by the `CadnoMorph` class. The first solid with function representation g_1 is represented as the parameter `zeroSolid`, indicating that this is the solid represented by the morph for a value of $t = 0$. Similarly, the second solid with function representation g_2 is parameter `oneSolid`. The floating point value of t is the `proportion` parameter. The bounding box for a morph is calculated by taking the bounding box for the set-theoretic union of the two defining point sets and then executing the `shrinkWrap()` method to approximate bounds just outside the morph solid shape.

Class Name	`CadnoMorph`	
Extends	`TopologyType`	

Value Format	`Morph` {
	    `zeroSolid` *FrepType*
	    `oneSolid` *FrepType*
	    `proportion` *float*
	}	

Parameters	Name	Type
	`zeroSolid`	`FrepType`
	`oneSolid`	`FrepType`
	`proportion`	`CadnoFloat`

Figure 7.13 shows a morph between a union of a box and a cylinder ("box/cylinder"), and a sphere with a cylindrical hole ("cut-sphere"). The figure is split into eight images with the corresponding value for t shown below each image. When $t = 0$, only shape for the box/cylinder is visible and when $t = 1$ then only the cut-sphere shape is visible. For values between 0 and 1, the shape can be observed to transform between the first and second defining solids. Two additional images are shown to demonstrate the effects that it is possible to achieve by using values for t that are outside the range $[0, 1]$.

A morph between two shapes can be defined implicitly in an *empirical world* script with the use of the `CadnoMakeMorph` class. The order for the arguments to the operator in a script is an optional `CadnoDetail` rendering level, followed by a floating point value for the `proportion` t, then the first shape (`zeroSolid`) with function representation g_1 and finally the second shape (`oneSolid`) with function representation g_2.

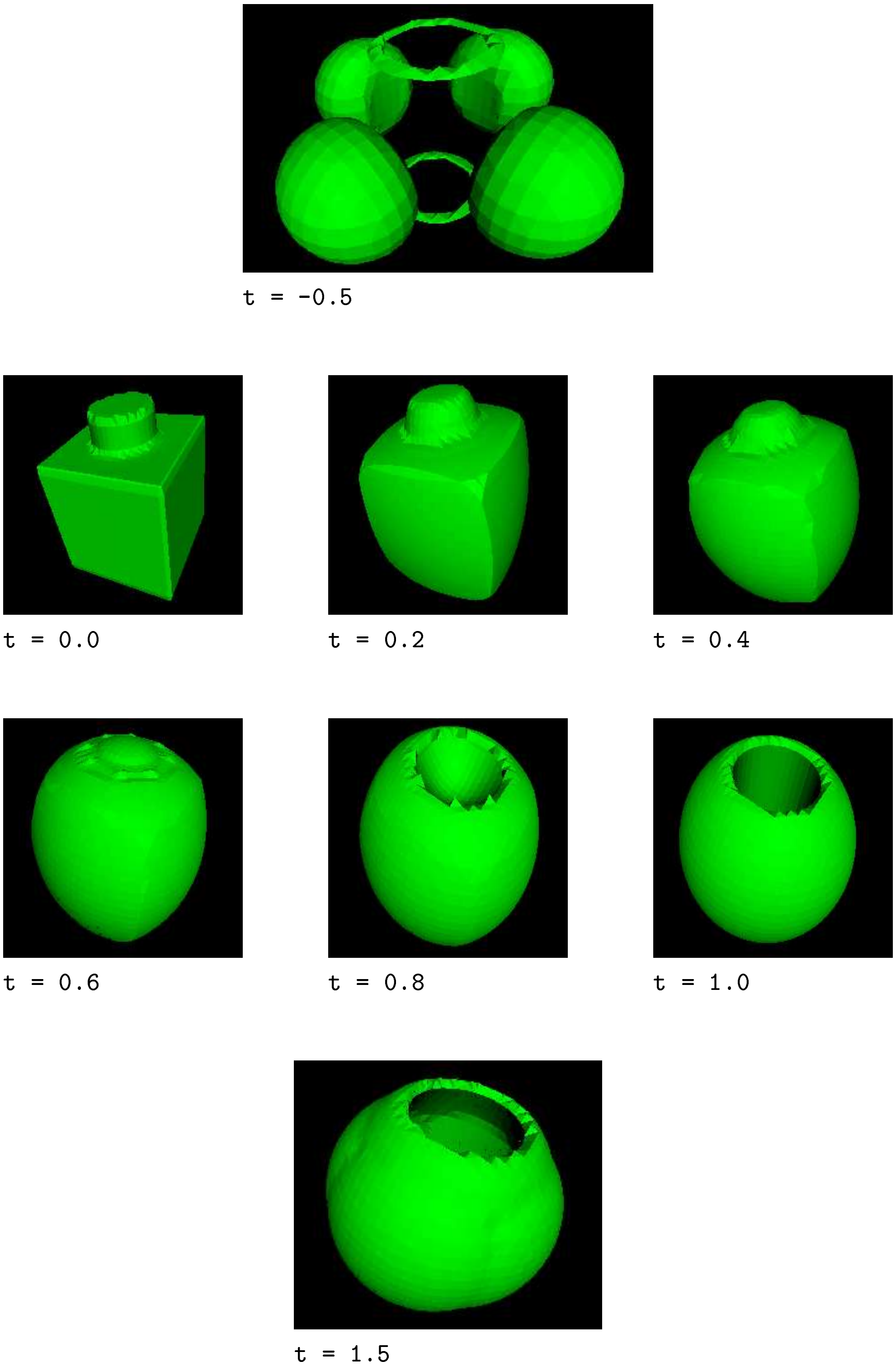


Figure 7.13: *Morph* between a box/cylinder shape and a cut-sphere shape, for varying values of `t`.

Operator Class		`CadnoMakeMorph`
Maps From		`CadnoDetail` × *float* × *FrepType* × *FrepType* or *float* × *FrepType* × *FrepType*
Maps To		`CadnoMorph`

Example	Defn.	`m1 = morph(0.5,`
		`Sphere { radius 1 },`
		`Cylinder { radius 1 height 2 } )`

	Value	`m1 = Morph {`
		`zeroSolid Sphere { radius 1 }`
		`oneSolid Cylinder {`
		`radius 1 height 2 }`
		`proportion 0.5`
		`}`

Chapter 8

Applications for Empirical Worlds

8.1 Introduction

In Chapter 7, *empirical worlds* and the associated Java class library are introduced. In this chapter, some applications of *empirical worlds* are described. They include:

- the integration of a broader range of shapes than represented by primitive shape data types. This is achieved by augmenting the *empirical world* class library. An example of this process is given in Section 8.2, which involves integrating operators and data types for the representation of *skeletal implicit surfaces* [WvO97, WvO95, BBCG$^+$97] into *empirical worlds*.

- the integration of a broader range of transformations than the standard affine transformations presented in Section 7.3.6. In Section 8.3, *warping* transformations for bending, twisting and tapering shapes are illustrated.

- a server/client implementation of a tool based on the *empirical world* classes with support for interaction using *empirical world* scripts. The *empirical world* *builder* server and world-wide-web client applications are described in Section 8.4.

- the creation of an artefact for the interactive exploration of a geometric shape using an *empirical world* script. A case-study is presented in Section 8.5 that illustrates the use of some of the data types and operators represented by the *empirical world* classes.

All the support for agency (see Section 6.3.2) and client/server implementation mechanisms (see Section 6.4.2) available in the JaM Machine API can be integrated into applications that are based on the *empirical world* class library. New interfaces to *empirical worlds* can be created by constructing appropriate graphical user-interfaces, or with parsers for a higher level geometric notations than *empirical world* scripts. The potential for real-time interaction with the *empirical world builder* tool is evaluated in Section 8.4.3, where timings for the polygonisation of shapes are presented.

8.2 Skeletal Implicit Shapes in Empirical Worlds

Skeletal implcits[1] are a class of implicit function shape constructed by blending sets of skeletal elements. Any skeletal element S, represented by the implicit mapping r_S, has the property that for a point $\mathbf{p}$, $r_S(\mathbf{p})$ is the minimum distance from $\mathbf{p}$ to the surface of S. The effect of this is that the values for r_S around a skeleton are like a contour map and define an increasing field as you move away from the skeletal geometry. The skeletons described in this section are sphere, cylinder and torus and all extend a class called `SoftType`. The skeletons described here are similar to those implemented by Larcombe in [Lar94] and could easily be extended in the future

[1] Also known as *Blobby* objects.

to include cone, ellipsoid, polygon and plane by implementing more subclasses of `SoftType`.

The approach adopted here is to integrate skeletal-based implicits into a modelling system that supports function representation. To do this the `SoftType` class is used, which directly extends `TopologyType` that in turn extends `FrepType` (see Figure 7.2). The `SoftType` class implements the function representation method `f()`. (see Section 7.2). In this section, it is assumed that if point $\mathbf{p}$ is inside the skeleton, then the value of $r_S(\mathbf{p})$ will be negative, this value is zero if $\mathbf{p}$ is on the surface of the skeleton and positive external to the skeleton. Instead of implementing method `f()`, all classes that extend `SoftType` implement another method called `d()` which returns the value of $r_S(\mathbf{p})$ for a particular skeleton. In the class `SoftType`, the method `f()` is implemented and returns the negative value mapped to by abstract method `d()`. The function representation f of a soft shape is $f(\mathbf{p}) = -r_S(\mathbf{p})$.

Wyvill and Guy suggest a syntax for skeletal implicit objects embedded as part of VRML in [WG97]. This syntax has been loosely observed in the implementation of the *empirical world* classes, except where mentioned in the text. The differences occur for the classes: `SoftShape`, which takes any skeleton S and applies a field function to the minimum distance value $r_S(\mathbf{p})$, for any point $\mathbf{p}$; `SoftSum`, which sums the fields surrounding a list of geometric objects. Both `SoftShape` and `SoftSum` have been extended so that they are effective for all instances of classes that extend `FrepType` in *empirical worlds*. More details are presented in Sections 8.2.4 and 8.2.5.

An important motivation for the inclusion of this section is to demonstrate how easily new JaM data types for a shape can be incorporated into an *empirical world* notation. The skeletal element types and operators represented (including the three skeletal elements described in this chapter, the `SoftShape` type, the `SoftSum` type and all the associated operators) were all implemented during one day. Once the classes themselves were written, it is a simple task to link them into a tool based on *empirical world* classes. As all the skeletal implicit classes extend `FrepType`, all operators available for the combination and manipulation of `FrepType` objects become immediately applicable to the soft objects, including affine transformations, CSG operations, morphing and so on. Conventional CSG-like geometric objects can coexist as components of models with *blobby* objects.

8.2.1 Soft Spheres

A soft sphere can be defined explicitly in *empirical world* scripts with the data type represented by the `CadnoSoftSphere` class. The sphere is parametrised by its `radius` r. Like the `CadnoSphere`, the `CadnoSoftSphere` is centred on the origin and should be translated to another position if a different centre is required by the user. The bounding box for the sphere has minimum point $(-r, -r, -r)$ and maximum point (r, r, r).

Class Name	`CadnoSoftSphere`	
Extends	`SoftType`	

Value Format	`SoftSphere` {
	   `radius` *float*
	}	

Default Value	`SoftSphere` {
	   `radius 1`
	}	

Parameters	Name	Type
	`radius`	`CadnoFloat`

A soft sphere can be defined implicitly in an *empirical world* script with the operator represented by the `CadnoMakeSoftSphere` class. The argument to the operator in an implicit definition for a soft sphere is the `radius` r.

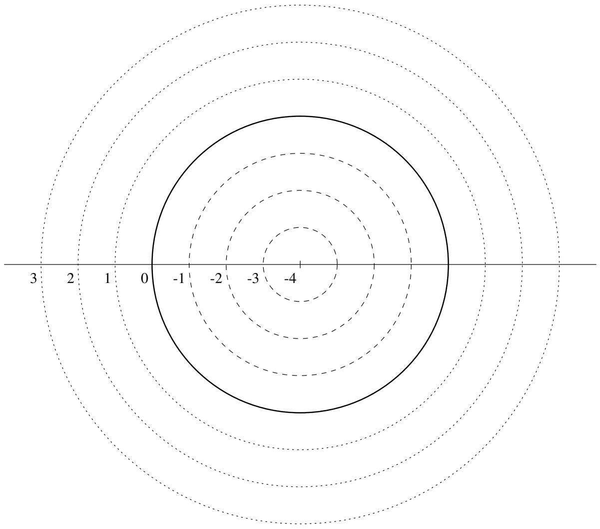


Figure 8.1: Planar slice through a `CadnoSoftSphere`, showing the surrounding field.

Operator Class		`CadnoMakeSoftSphere`
Maps From		`CadnoDetail` × *float* or *float*
Maps To		`CadnoSoftSphere`
Example	Defn.	`ss = softSphere(3.2)`

	Value	`ss = SoftSphere {`
		`radius 3.2`
		`}`

The distance d from any point (x, y, z) to the skeleton surface of the soft sphere with radius r is given by

$$d(x, y, z) = \sqrt{x^2 + y^2 + z^2} - r \tag{8.1}$$

This function is C^1 continuous everywhere except at the origin point (0, 0, 0).

Figure 8.1 shows the field around and inside a sphere of radius 4 given by d for a planar slice through the solid sphere geometry in the xy-plane. The thick line represents the surface of the sphere, the dashed lines indicate the field inside the sphere and the dotted lines indicate the field outside the sphere.

8.2.2 Soft Cylinders

Soft cylinders are parametrised in *empirical worlds* by a height h and a radius r and the geometric shape represented has hemispherical ends, simplifying the calculation of the distance field. A `CadnoSoftRCylinder` class can be used to explicitly define the value of a soft cylinder, centred on the origin with its height dimension oriented along the y-axis of the space. The parameters of a cylinder in a script are `height` and `radius`. The bounding box for soft cylinder is defined by minimum point $(-r, \frac{-h}{2} - r, -r)$ and $(r, \frac{h}{2} + r, r)$.

<table>
<tr><td>Class Name</td><td colspan="2"><code>CadnoSoftRCylinder</code></td></tr>
<tr><td>Extends</td><td colspan="2"><code>SoftType</code></td></tr>
<tr><td>Value Format</td><td colspan="2"><code>SoftRCylinder</code> {
    <code>radius</code> float
    <code>height</code> float
}</td></tr>
<tr><td>Default Value</td><td colspan="2"><code>SoftRCylinder</code> {
    <code>radius 1</code>
    <code>height 2</code>
}</td></tr>
<tr><td>Parameters</td><td>Name</td><td>Type</td></tr>
<tr><td></td><td><code>height</code>
<code>radius</code></td><td><code>CadnoFloat</code>
<code>CadnoFloat</code></td></tr>
</table>

Soft cylinders can be defined implicitly in *empirical world* scripts with the operator represented by the `CadnoMakeSoftRCylinder` class, which extends `DefnFunc`. The arguments to the operator in an implicit definition are a value representing the `radius` r of the soft cylinder, followed by the `height` h of the cylinder.

Operator Class		`CadnoMakeSoftRCylinder`
Maps From		`CadnoDetail` × *float* × *float* or *float* × *float*
Maps To		`CadnoSoftRCylinder`
Example	Defn.	`sc = softRCylinder(3.1, 7.9)`

	Value	`sc = SoftRCylinder {`
		`        radius 3.1`
		`        height 7.9`
		`        }`

The distance d from any point (x, y, z) to the surface of the skeleton of a soft cylinder is given by

$$d(x,y,z) = \begin{cases} \sqrt{x^2 + (y - \frac{h}{2})^2 + z^2} - r & \text{if } y \geq \frac{h}{2} \\ \sqrt{x^2 + z^2} - r & \text{if } -\frac{h}{2} < y < \frac{h}{2} \\ \sqrt{x^2 + (y + \frac{h}{2})^2 + z^2} - r & \text{if } y \leq -\frac{h}{2} \end{cases} \tag{8.2}$$

This mapping is C^1 continuous everywhere except at the planes $y = -\frac{h}{2}$ and $y = \frac{h}{2}$ and along the y-axis between these planes.

Figure 8.2 shows the field inside and outside a soft cylinder as given by the mapping d for a two-dimensional slice through the solid geometry in the xy-plane.

8.2.3 Soft Tori

One way that a torus shape can be parametrised is by an inside radius ir and an outside radius or. In an *empirical world* script, a soft torus is centred on the origin and inside and outside radius are measured in the xz-plane. The inside radius ir is the parameter `insideRadius` in a script and outside radius or is the parameter `outsideRadius`. The bounding box for a torus is defined by minimum point $(-or, -\frac{or-ir}{2}, -or)$ and maximum point $(or, \frac{or-ir}{2}, or)$.

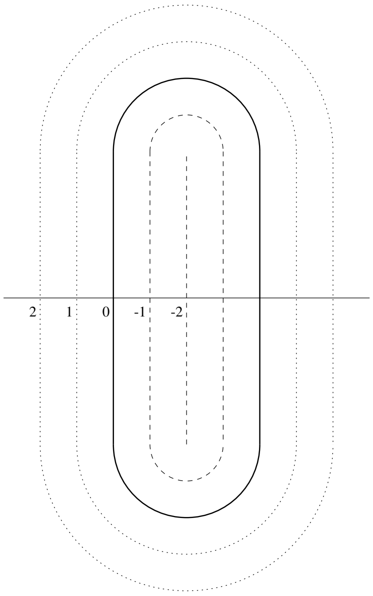


Figure 8.2: Planar slice through a `CadnoSoftRCylinder`, showing the surrounding field.

Class Name	`CadnoSoftTorus`	
Extends	`SoftType`	

Value Format	`SoftTorus` {
	    `insideRadius` *float*
	    `outsideRadius` *float*
	}	

Default Value	`SoftTorus` {
	    `insideRadius 0.5`
	    `outsideRadius 1.0`
	}	

Parameters	Name	Type
	`insideRadius`	`CadnoFloat`
	`outsideRadius`	`CadnoFloat`

Soft tori can be implicitly defined in *empirical world* scripts using the operator represented by the `CadnoMakeSoftTorus` class. The arguments to this operator in an implicit definition of a torus shape are the value of the inner radius `insideRadius` ir, followed by the value of`outsideRadius` or.

Operator Class		`CadnoMakeSoftTorus`
Maps From		`CadnoDetail` × *float* × *float* *float* × *float*
Maps To		`CadnoSoftTorus`

Example	Defn.
	Value
		`st = softTorus(0.3, 0.8)`
		`st = SoftTorus {`
		`insideRadius 0.3`
		`outsideRadius 0.8`
		`}`

The distance field function d for any point (x, y, z) for a soft torus with inner radius ir and outer radius or is

$$s(x,y,z) = \sqrt{x^2+z^2} - \left(ir + \frac{or-ir}{2}\right) \quad (8.3)$$

$$d(x,y,z) = \sqrt{s(x,y,z)^2+y^2} - \frac{or-ir}{2} \quad (8.4)$$

Every point defines a plane that embeds the point and the y-axis. The distance from the skeletal element is the function representation for a two-dimensional solid circle embedded in this same plane with its centre point located on the xz-plane at a distance of $ir + \frac{or-ir}{2}$ away from the origin. This function is C^1 discontinuous at the origin point $(0, 0, 0)$ and on the circle in the xz-plane defined by $s(x, 0, z) = 0$ and $y = 0$.

Figure 8.3 shows the distance field inside and outside a soft torus for a torus with an inner radius of 3 and an outer radius of 5. The field shown is a planar slice through the solid torus in the xz-plane.

8.2.4 Field Functions and Soft Shapes

A field function can be applied to an object of `SoftType` to modify the linearly increasing fields around soft objects, as illustrated in Figures 8.1, 8.2 and 8.3. The contribution of a component shape should only have a significant effect on the whole

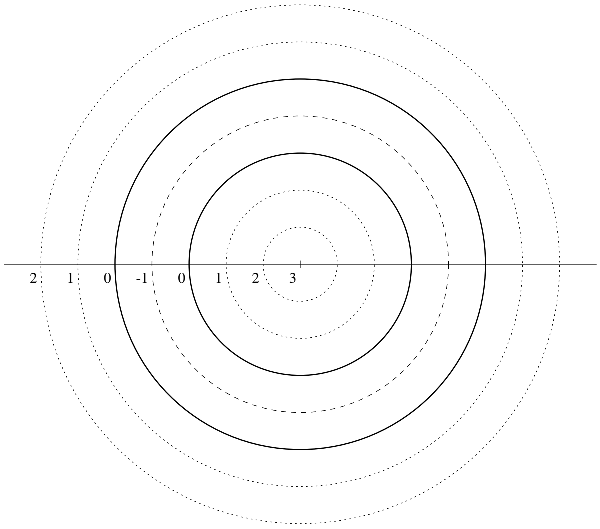


Figure 8.3: Planar slice through a `CadnoSoftTorus`, showing the surrounding field.

shape when components are in close proximity to one another. When skeletal elements are combined by summing the contribution of their function to the field, that contribution cannot be a linear distance fuction as two objects located a significant distance away from one another can interfere with one another. Field functions, typically a cubic function applied to the linear distance fuction for a point, allow a component to have its maximum effect on the whole shape close to its boundary.

In *empirical world* scripts, the field function has been extended so that it also operates for fields around any function representation of geometric shape. Modifying the field around a skeleton effects how geometric objects interact in blends, morphs, sums and so on. For geometric objects that extend `SoftType`, this is a well defined field with values dependent on distance away from the skeleton. For solid geometry represented by classes that do not extend `SoftType`, a field may not be uniform and a user should be aware of this in their selection of a field function. After the application of the field function, any distance properties of the fucntion representation are significantly reduced or lost and the field function representation produces poor results when used in blending and morphing operators, in comparison to the original representation.

To explicitly define a soft shape in an *empirical world* script, a user can use the `CadnoSoftShape` class, which extends `TopologyType`. The parameters required for this explicit definition are:

- the existing geometric shape given by function representation g that is considered a `skeleton` shape;
- a field function parameter is required to define how the field varies for a particular value of g at any point. The parameter is called the `fieldFunction` and must be one of the explicit definitions for a `GraphType` (see Appendix A.5).
- a `weight` value w by which the value of the field function of the skeleton function at a particular point is multiplied;
- the `isoValue` i which determines at what level the value returned by the weighted field function is considered to be the surface of the geometry.

The bounding box for a `CadnoSoftShape` is calculated by starting with the bounding box for the `skeleton` geometry and then executing the `shrinkWrap()` method. The resulting shape for certain weight values w, field functions h or isovalues i may be infinite in which case the bounding box will clip or even miss altogether the bounds of the shape. The user of an *empirical world* script should be aware that this is a possible problem and should choose their parameters carefully.

Class Name	`CadnoSoftShape`	
Extends	`TopologyType`	

Value Format	`SoftShape` {
	    `skeleton` *FrepType*
	    `fieldFunction` *GraphType*
	    `weight` *float*
	    `isoValue` *float*
	}	

Parameters	Name	Type
	`skeleton`	`FrepType`
	`fieldFunction`	`GraphType`
	`weight`	`CadnoFloat`
	`isoValue`	`CadnoFloat`

Soft field functions can be defined implicitly in *empirical world* scripts by using the `CadnoMakeSoftShape` class, which extends `DefnFunc`. The arguments to this operator in an implicit definition of a soft shape are:

1. the `skeleton` geometric object of `FrepType`;
2. the `fieldFunction` instance of a class that extends `GraphType`;
3. the `weight` value w;
4. the `isoValue` i.

Operator Class		`CadnoMakeSoftShape`
Maps From		`CadnoDetail` × *FrepType* × *GraphType* × *float* × *float* or *FrepType* × *GraphType* × *float* × *float*
Maps To		`CadnoSoftShape`
Example	Defn.	`ss = softShape(SoftSphere { }, FieldGraph { }, 1.1, 0.2)`
	Value	`ss = SoftShape { skeleton SoftSphere { radius 1 } fieldFunction FieldGraph { raise 2 constant 1 coefficient 2 } weight 1.1 isoValue 0.2 }`

Field functions for use in modelling with skeletal implicits are intended to work for soft objects where the argument to the function is a positive distance away from the skeleton if the point of sampling is outside the skeletal geometry. To remain consistent with this, the argument to the field function in a `CadnoSoftShape` is a negative value of original function representation g for any point (x, y, z). If g is a function representation for one of the geometric objects of `SoftType`, this achieves consistency because for any instance of `SoftType`, $g(x, y, z) = -d(x, y, z)$.

The function representation f for a soft shape at any point (x, y, z) is

$$f(x, y, z) = i - wh(-g(x, y, z)) \tag{8.5}$$

It is not possible to make any claim about the continuity of this function because the continuity of the field function h is not known. To convert the value calculated for the field function into a function representation for the geometric shape, this value is subtracted from the isovalue i. The `isovalue` parameter allows a user to interactively experiment so as to determine which level of the field produces the most suitable shape and size of geometry for their purposes.

8.2.5 Summing the Fields of Skeletal Implicit Shapes

Once definitions of skeletal geometry exist as function representations, all the set theoretic operations, affine transformations, morphing and warps can be used to modify and combine their shape with other geometric shapes, be these soft or otherwise. One special feature of skeletal geometry is that it is combined with appropriate field functions, such as the `FieldGraph` available in *empirical world* classes (see Appendix A.5), field values at any point can be summed so that when two soft shapes are in close proximity to each other they appear to *automatically* blend. This effect can be achieved in *empirical world* scripts using the explicit data type of the `CadnoSoftSum` class.

The parameters to a `CadnoSoftSum` class are a list of `children` of function representations for geometry, given by $g_1, \ldots, g_n$. The other parameter to an explicit definition of a value of this type is the field function sampling level i called `isoValue`.

Class Name	`CadnoSoftSum`	
Extends	`TopologyType`	

Value Format	`SoftSum {`
	    `isoValue` *float*
	    `children` [(*FrepType*)*]
	`}`	

Parameters	Name	Type
	`isoValue`	`CadnoFloat`
	`children`	List of objects of `FrepType`.

Soft sum geometric shape can be implicitly defined in an *empirical world* script with the use of the `CadnoMakeSoftSum` class, which extends `DefnFunc`. The arguments to the operator in an implicit definition for a soft shape are the isovalue i followed a comma separated list of geometric objects with function representation.

These geometric objects should have an appropriate field function applied to their original skeletal defining geometry, such as the `FieldGraph`, and the soft sum is likely to produce the best results for geometry with skeletons that extend `SoftType`.

Operator Class		`CadnoMakeSoftSum`
Maps From		`CadnoDetail` × *float* × (*FrepType*)* or *float* × (*FrepType*)*
Maps To		`CadnoSoftSum`
Example	Defn.	`ss = softSum(0.1, ssh1, ssh2)` *ssh1* and *ssh2* should be `CadnoSoftShapes`

	Value	`ss = SoftSum {`
		`isoValue 0.1`
		`children [ ssh1 ssh2 ]`
		`}`

The function representation f for a soft sum at any point (x, y, z), the sum of all the fields of its `children`, is given by the equation

$$f(x, y, z) = i - \sum_{j=1}^{j=n}(-g_j(x, y, z)) \tag{8.6}$$

Figure 8.4 illustrates soft summing of a soft torus centred at the origin and a soft sphere that is translated by the vector given below each image. As the sphere is translated towards the torus, the fields of the two objects merge, creating the effect of the geometric objects blending. The isovalue for the sum shown was 0.5, and the field function h for any value x is given by the equation shown in Figure 8.5 and given by the equation

$$h(x) = 2(1 - 2^{-x})2^{-x} \tag{8.7}$$

8.3 Warping Transformations in *Empirical Worlds*.

The transformations represented by a `CadnoTransform` class are the common affine transformations (see Section 7.3.6). In *empirical worlds*, some interesting effects can

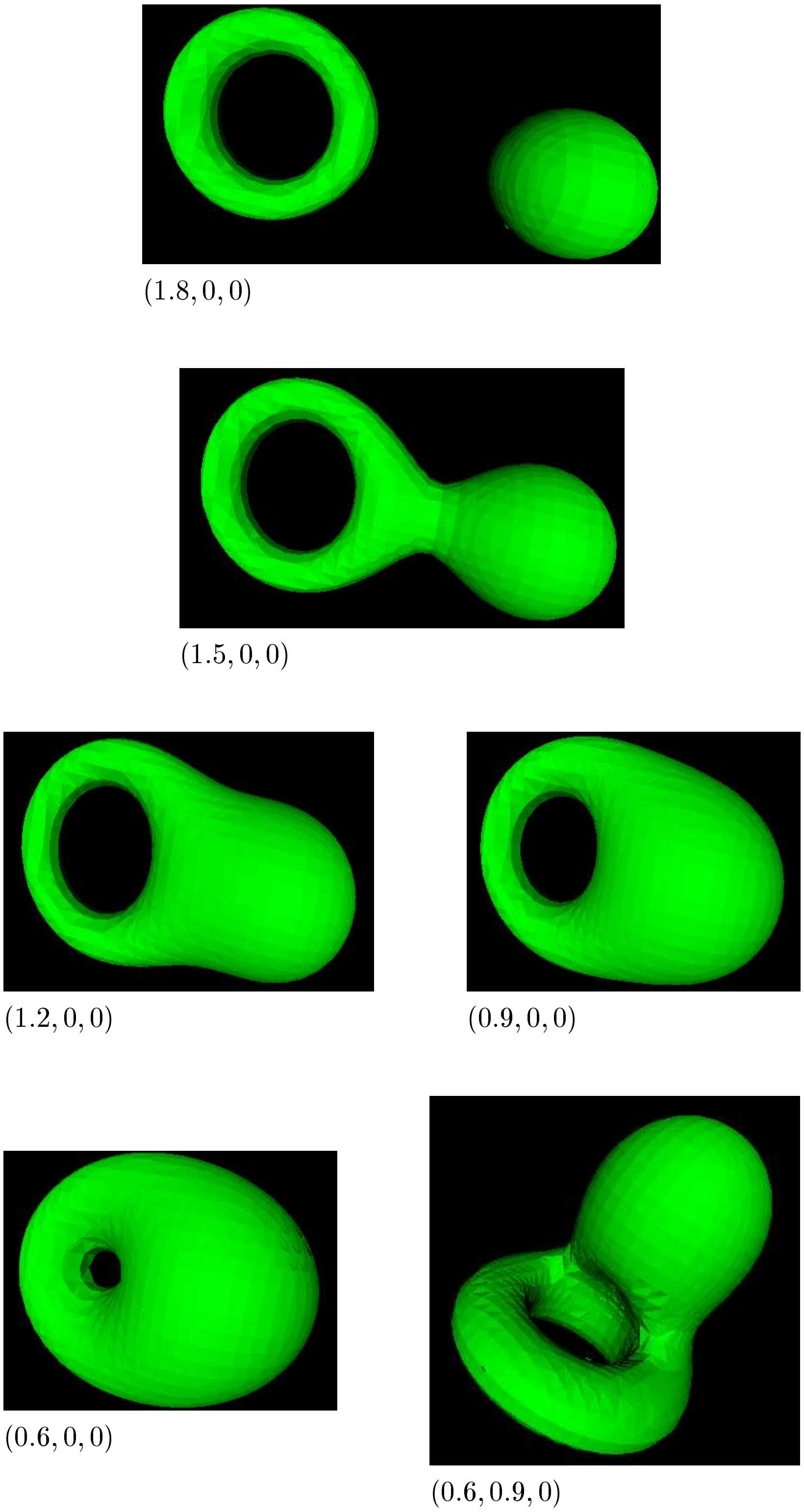


Figure 8.4: Images of a `CadnoSoftSum` of a `CadnoSoftTorus` and a translated `CadnoSoftSphere` (translation vector shown by each image).

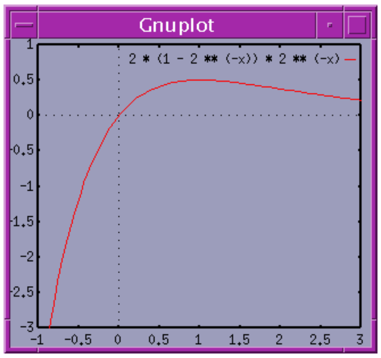


Figure 8.5: Plot of example field function h

be achieved through other transformations (linear and non-linear) that simulate the effect of tapering, bending and twisting objects.

Each of the transformations described in this section is based on the operations described by Wyvill and van Overveld in [WvO97] which are based in turn based on those described by Barr in [Bar84]. For any point $\mathbf{p}$ in space and function representation of a shape g, the warping transformation w of the shape at point $\mathbf{p}$ is a mapping from warped space to Euclidean space such that

$$f(\mathbf{p}) = g(w(\mathbf{p})) \tag{8.8}$$

In this formula, f is the function representation of the warped shape. In *empirical worlds*, each warp has a type that extends `DefnType` and an operator for their implicit definition that extends `DefnFunc`. Each type or operator takes one existing (`original`) piece of geometry and performs the transformation on this geometry. In this section, new VRML-like syntax is introduced for the description of values for data types represented.

8.3.1 Linear Taper

A linear taper operation has a defining axis and a dimension that it affects. In the warped space, the affected dimension has every coordinate point value increased by a factor in linear proportion to the point's projected coordinate value along the taper's defining axis. The example of a linear taper in *empirical world* scripts is a taper of the x dimension along the z defining axis. If the axis and defining dimension are not appropriate, then the geometry can be transformed to suit the taper operation and then transformed back to its original location.

The data type for explicit definition of an instance of a tapered geometric object in an *empirical world* scripts is represented by the `CadnoTaperXZ` class. Explicit definitions requires two parameters:

- a `taperRate` w that represents the rate at which the warp affects the coordinates of the x dimension proportional to the z coordinate;
- an explicit value description of the `original` geometry prior to the warp. This pre-warp geometry can be any geometric object given by a function representation.

The bounding box for the newly tapered shape is calculated by tapering the bounding box for the original geometry and finding a bounding box for this.

Class Name	`CadnoTaperXZ`	
Extends	`TopologyType`	

Value Format	`TaperXZ {`
	    `taperRate` *float*
	    `original` *FrepType*
	`}`	

Parameters	Name	Type
	`taperRate`	`CadnoFloat`
	`original`	`FrepType`

A linearly tapered shape with a defining axis along z and effected dimension x can be implicitly defined in an *empirical world* script by using the `CadnoMakeTaperXZ` class, which extends `DefnFunc`. The arguments in an implicit definition of a taper are the `taperRate` parameter w followed by the pre-warp `original` geometric shape with function representation g.

Operator Class		`CadnoMakeTaperXZ`
Maps From		`CadnoDetail` $\times$ *float* $\times$ *FrepType* or *float* $\times$ *FrepType*
Maps To		`CadnoTaperXZ`
Example	Defn.	`tp = taperXZ(-0.4, Box { })`

	Value	`tp = TaperXZ {`
		`taperRate -0.4`
		`original Box { size 2 2 2 }`
		`}`

The function representation f for a tapered shape as defined in an *empirical world* script by a definition of a `CadnoTaperXZ` is given by the function f below. It is obvious from the equation that coordinates in the y and z dimension are preserved and only the x dimension is effected. An example of such a taper is shown in Figure 8.14 where the original geometry is shown in Figure 8.12.

$$f(x, y, z) = g(x(1 + zw), y, z)$$

8.3.2 Bend

In a similar way to the taper operation, a bending operation has a defining major axis and a dimension along which its effect is measured. An analogy to real-world bending is that the defining axis is like a fixed post about which the geometry is to be bent by applying a force at each end of the geometry. This is illustrated in Figure 8.6 that shows three stages of bending a long box about the major defining z-axis, where the cross "$\times$" represents the origin point of the space. As a force is applied to both ends of the geometry, each point of the geometry bends by an angle

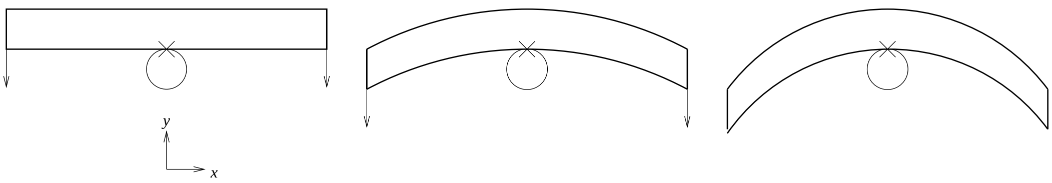


Figure 8.6: Planar slice in the xy-plane showing the bending of a long box about the z-axis with the amount of bend measured along the x-axis.

proportional to the the x component of its original point.

In virtual spaces, like those defined in an *empirical world* script, a bend operation can be achieved by specifying a bend rate k in radians per unit length and an offset value for the centre of the bend along the x-axis *x0*. An explicit definition of some bent geometry about the z-axis with amount of bend measured along the x-axis is represented using the `CadnoBendXZ` class. An explicit definition contains the explicit values for a floating point parameter called `bendRate` which is the value for k, a floating point value `offset` which is a value for *x0* and `original` which is an explicit description of a geometric shape that is to be bent by this operator, with function representation g. The bounding box for the bent geometry is found by taking the bounding box for the original geometry and executing the `shrinkWrap()` method.

Class Name	`CadnoBendXZ`	
Extends	`TopologyType`	

Value Format	`BendXZ` {
	 `offset` *float*
	 `bendRate` *float*
	 `original` *FrepType*
	 }	

Parameters	Name	Type
	`offset`	`CadnoFloat`
	`bendRate`	`CadnoFloat`
	`original`	`FrepType`

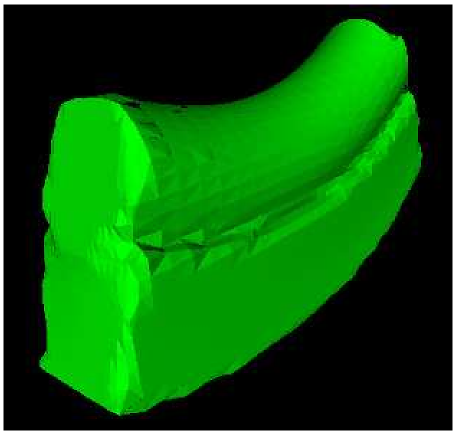


Figure 8.7: Image of a blended long box and cylinder that have been bent around the z-axis, along the x-axis.

Figure 8.7 shows a blended long box and cylinder that have then been bent by the bending operation.

Bent geometric shapes, with major defining axis the z-axis and with the amount of bending measured along the x-axis, can also be defined implicitly in *empirical world* scripts using the `CadnoMakeBendXZ` class, which extends `DefnFunc`. The order of arguments in an implicit definition of a bend are the `bendRate` value k, followed by the `offset` parameter *x0* and the `original` solid geometry with function representation g.

Operator Class		`CadnoMakeBendXZ`
Maps From		`CadnoDetail` $\times$ *float* $\times$ *float* $\times$ *FrepType* or *float* $\times$ *float* $\times$ *FrepType*
Maps To		`CadnoBendXZ`
Example	Defn.	`bd = bendXZ(1, 0.78, Box { size 5 1 2 })`

	Value	`bd = BendXZ {`
		`offset 1.0`
		`bendRate 0.78`
		`original Box { size 5 1 2 }`
		`}`

The function representation for the bent geometry f at any point (x, y, z) is

$$f\begin{pmatrix} x \\ y \\ z \end{pmatrix} = g\begin{pmatrix} -\sin(k(x-x0))(y-\frac{1}{k})+x0 \\ \cos(k(x-x0))(y-\frac{1}{k})+\frac{1}{k} \\ z \end{pmatrix} \tag{8.9}$$

The warping has no effect on the z dimension and it can be seen in the equation how the amount of bend is proportional to the value of the offset of the x coordinate of a point.

8.3.3 Twist

A twist operation is can be defined to operate around one major axis. A twisted piece of geometry is created from its original geometry by rotating each point in the original geometry around the defining axis through an angle proportional to the component distance for the point along the defining axis. In *empirical world* scripts, a twist of geometry along the z-axis can be explicitly defined using the `CadnoTwistZ` class. Parameters for twisted geometry are a floating point value for the `twistRate` w in radians per unit length measured along the z-axis and the `original` solid geometry with function representation g. The bounding box for such geometry is found by taking the bounding box for the `original` geometry and executing the `shrinkWrap()` method on the twisted version of the geometry to find the new bounds.

Class Name	`CadnoTwistZ`	
Extends	`TopologyType`	

Value Format	`TwistZ {`
	    `twistRate` *float*
	    `original` *FrepType*
	`}`	

Parameters	Name	Type
	`twistRate`	`CadnoFloat`
	`original`	`FrepType`

Figure 8.8 shows a blended box and cylinder twisted around the z-axis in

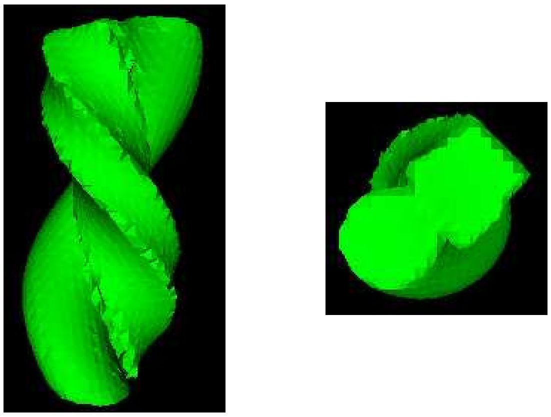


Figure 8.8: Image of a blended box and cylinder twisted around the z-axis

.

side and top views.

Twisted geometric shapes, with defining axis the z-axis, can be defined implicitly in *empirical world* scripts using the `CadnoMakeTwistZ` class, which extends `DefnFunc`. The arguments to an implicit definition of a twist are the `twistRate` w followed by the `original` pre-warp geometry with function representation g.

Operator Class		`CadnoMakeTwistZ`
Maps From		`CadnoDetail` × *float* × *FrepType* or *float* × *FrepType*
Maps To		`CadnoTwistZ`
Example	Defn.	`tw = twistZ(detail 10 10 30, 0.78,` `Box { size 1 2 5 })`

	Value	`tw = TwistZ {`
		`twistRate 0.78`
		`original Box { size 1 2 5 }`
		`}`

The function representation f at any point (x, y, z) for a geometric shape twisted around the z-axis is

$$f\begin{pmatrix} x \\ y \\ z \end{pmatrix} = g\begin{pmatrix} x\cos(zw) - y\sin(zw) \\ x\sin(zw) + y\cos(zw) \\ z \end{pmatrix} \tag{8.10}$$

The warping transformation preserves the z-coordinate component of points and rotates the x and y components by an angle proportional to the z component.

8.4 Implementation of Empirical Worlds

The *empirical world builder* tool is an application based on the empirical world classes illustrated this chapter and Chapter 7. The application is a computer-based tool for exploration of shape in an empirical manner. The *empirical world builder* tool is based on a server/client architecture, where a computer acting as a world-wide-web or file server also runs the server-side application. This application is a *JaM server* as described in Section 6.4.2 of Chapter 6. The client application, a *JaM Client*, connects to the server system through network sockets and *empirical world* scripts can be created and edited through the client. The client also includes a VRML browser that allows a user to interactively explore the shape they have described in their *empirical world* script.

8.4.1 Empirical World Builder Server

The server, called `EWServer`, is a command line based Java application that can be run on any Java Virtual Machine. The server is multi-threaded application that listens for connections on a TCP/IP network socket. When the server receives a connection request on the port to which it is listening, a new `java.lang.Thread` class is created to handle the connection. No limit is set for the number of clients that can connect to the server, although there are some system specific limits on numbers of connections that can restrict the numbers of simultaneous clients.

When a new thread is created, its initialization creates an instance of the `JaM.Script` class and adds all the *empirical world* types and functions to this `Script`. There is only one agent for each of these scripts, the `root` super-user agent[2]. The thread then enters a continuous loop which waits at the start of each iteration for user input from the associated client. User input can include new definitions, requests for information on current definitions and instruction to close the network socket and shutdown the thread. All messages sent from the client to the server are streams of characters that are converted to `java.lang.String` objects on the server. Messages from the server to the client are either streams of characters for printing in the client's output window, or streams of data representing some VRML for display in the client's VRML browser.

In response to new definitions or redefinitions, the `Script.addToQ()` method is called to add these definitions to the queue of definitions waiting for the next update of the `Script`. In the strings of characters from a client, the server recognises new definitions or redefinitions as they contain an equals sign (`=`). If the `addToQ()` or `update()` method for the script instance throws an exception due to recently received definitions, an error message is sent to the client along with the `String` of characters that triggered the exception. This allows the user to re-edit the string in case the reason for the error was a simple typing mistake.

In addition to definitions, direct instructions can be issued from the client to the server. A user of the client can type the following instructions:

`update` Causes the server to call the `Script.update()` method so that if there are any redefinitions on the queue, the state of the script is updated.

`value` *id* A user instructs the server to return the value of identifier *id* in the current *empirical world* script, by calling the `printVal()` method (see Table 6.5). The server either returns the explicit value of *id*, or an error if there is no identifier called *id* in the script.

`defn` *id* A user instructs the server to return the implicit definition of the identifier

[2]See Section 6.3.2 in Chapter 6 for more details.

id in the current *empirical world* script by calling the `printDef()` method. The server either returns the implicit definition of *id* if it is implicitly defined, otherwise the explicit value is returned. If there is no identifier called *id* in the script, an error is reported to the client.

`printall` A user request from the client that instructs the server to send to the client a list of all definitions in the current state of the *empirical world* script. The server does this by creating a string of characters containing the output of the method `printAll()` that contains every definition in the script.

`?` *id* A user instructs the server to return to the client some additional information about identifier *id* other than its explicit value or implicit definition. The server calls the `inspectDef()` method of the script instance for the identifier *id*. If *id* exists within the script, its owner, permission fields and dependents definitions are sent back in `String` form to the client, otherwise an error is reported. In the current implementation, JaM permissions and owners are not in use so every definition is owned by the super-user `root` and has all definition permissions set to "`rwr-`".

`exit` A user instructs the server to close the network connection to the client and destroy the thread for handling this connection. The destruction of the thread also destroys the *empirical world* script that was associated with the connection to the client.

The server also performs any necessary polygonisation of implicit shapes in classes that extend the `TopologyType` class. This polygonisation is described in Section 8.4.3 of this chapter. The motivation for polygonisation to be carried out on the server is the assumption that a server computer often has more numerical computing power than a client and client computers resources can be dedicated to the exploration of shape through a VRML browser.

8.4.2 Empirical World Builder Client

The *empirical world builder* client application is implemented as a web page containing a *CosmoPlayer*[3] VRML browser plugin and a Java applet that communicates with the VRML browser and the *empirical world* server. In the screen snapshot of a Netscape Communicator browser shown in Figure 8.9, the client web page is demonstrated when in use. The top panel in the window with some geometry displayed and some viewing controls is the CosmoPlayer browser. The bottom panel containing two text components and two buttons is the Java applet interface between the VRML browser and the *empirical world* server.

The Java applet in the bottom panel is initially disconnected from the server. The two buttons are labelled `Connect` and `Update`. Clicking on the `Connect` button causes the client to try to connect to an *empirical world builder* server on the server from which the whole web page was downloaded[4]. Output from the server and error messages directly from the client appear in the top text component of the applet panel, which has a grey background. The user cannot edit any text in the component but can copy its contents to the operating system clipboard. When connection to the server is established, the message "`Connected to server.`" appears in this output window and the `Connect` button changes to a "`Break`" button that forces disconnection.

One implementation special operator exists in *empirical world* scripts that defines the interface between the values for definitions in a script and the client's VRML

[3]This web browser plugin, developed by a subsidiary of Silicon Graphics Inc., is freely available and runs under Windows 95, Windows NT, HP-UX and Irix. See `http://www.cosmoplayer.com/` for more information.

[4]One of the security restrictions enforced by Java applet *Security Managers* is that a network client applet can only connect to the server from which the Java classes for that applet were downloaded [Har97].

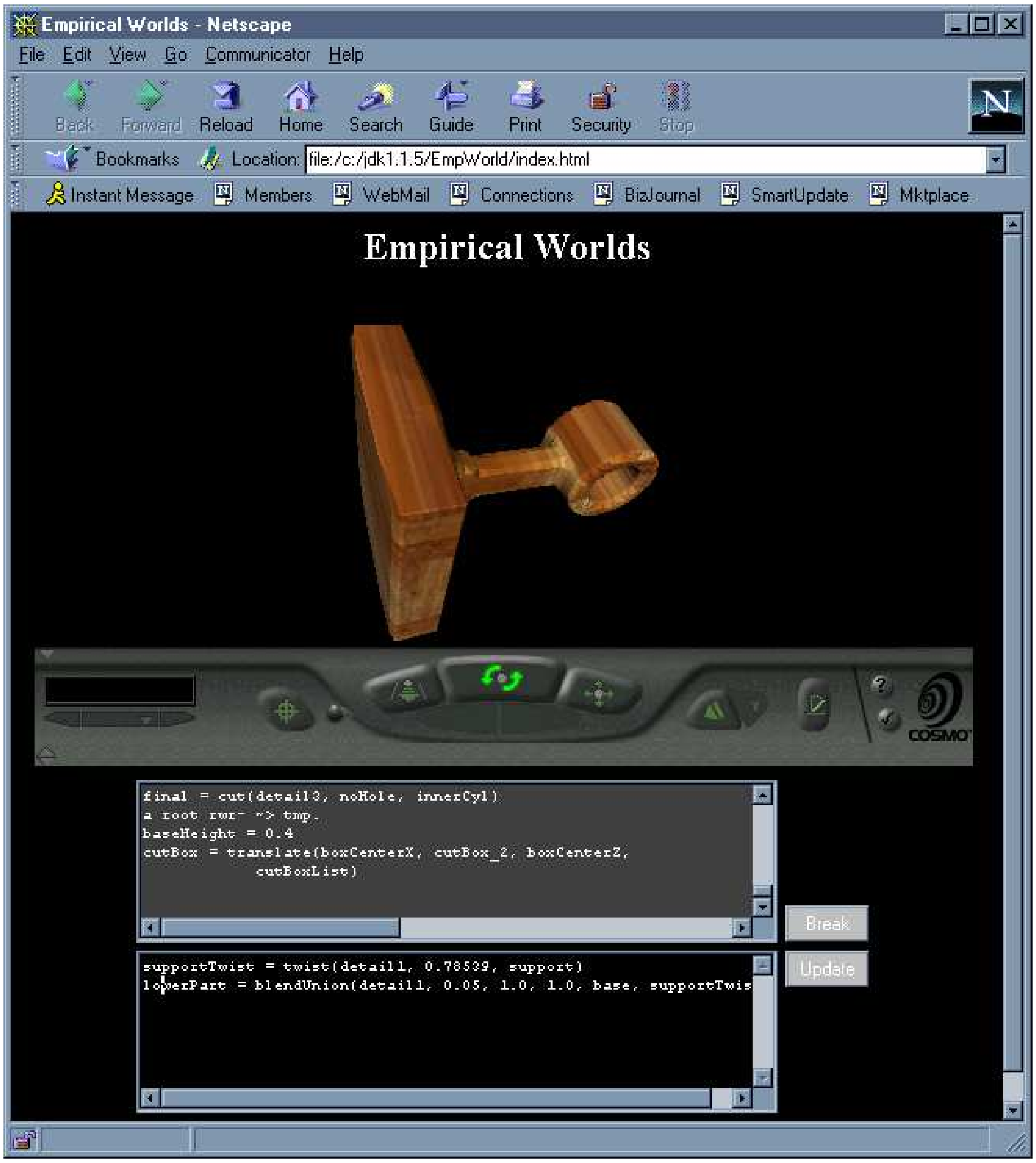


Figure 8.9: Screen snapshot of a web browser when viewing the interactive *empirical worlds* page.

browser or a file. This is the `CadnoWorld` class that directly extends `DefnFunc`[5]. In its current form, the operator maps a list of geometric objects (`FrepList`) defined by function representations to the same list, with the side effect that a file containing VRML-2 nodes to describe that geometry is generated. For an implicit definition of the form "*identifier* `= world (`*list1*`)`", a file called "*identifier*`.wrl`" is generated on the local filing system to the server, in the same directory from when the *empirical world* client was downloaded. A special case exists when the *identifier* is "`w`". Instead of the VRML being sent to a file, it is sent to the VRML browser of the *empirical world* client[6].

Operator Class		`CadnoWorld`
Maps From		`FrepList`
Maps To		`FrepList`
Example	Defn.	`w1 = world(children [ Box { } ])`
	Value	`w1 = children [ Box { } ]`
		Generates VRML file "`w1.wrl`".

The user types new input into the bottom text component of the applet panel, which has a black background. Each line of the text is considered as a separate definition, redefinition or server instruction. When the `Update` button is pressed, the current content of the text component is sent to the server line by line and the server responds in the way described in Section 8.4.1. The client appends an `update` instruction onto the end of each package of definitions and instructions, so that the script on the server will be updated consistent with any definitions currently on the scripts queue every time the `Update` button is clicked. The input text component is also cleared, awaiting further new user input.

[5]The position of the class in the *empirical world* class hierarchy is shown in a circled box in the class diagram in Figure 7.2.

[6]The variable `w` in *empirical world builder* has the same purpose as the `screen` variable in SCOUT [Dep92].

If there are any errors during the `Script.addToQ()` method on the server then these are reported by being appended to the end of the output text component. The definition which caused the error is returned to the client and placed in the input window for editing and then resubmission to the server by the user.

If the update propagate to the special *world* identifier "`w`" then the VRML displayed in the CosmoPlayer VRML browser is updated consistent with the definition of "`w`". The user can explore the selected redefined shapes which are dependencies for `w`, after its VRML description has been transmitted from the server to the client and then rendered by the client in the browser. This process normally takes no more than a few seconds. The user can then rotate, zoom, pan, seek sections, fly around and walk around the geometry displayed in the VRML browser.

In the current implementation, there is no facility for file save and load for an *empirical world* script. It is possible to cut and paste between the text components of the client's applet and other applications such as text editors. To save the current state of a script, a user can issue the `printall` command and retrieve a list of all definitions and their current values in the output text component. This can be selected and pasted into a text editor, edited and saved to a file. Files of definitions can be loaded into the client by selecting and copying to the clipboard the required definitions and then pasting them into the input window.

8.4.3 Polygonisation of Implicit Solid Geometry

The polygonisation of implicit shape used for the *empirical world builder* tool is simplistic and does not produce the best visual results. It was written to be fast and simple, to demonstrate that the tool has potential as an interactive modelling tool rather than to produce high quality visual images suitable for brochures or publications. The polygonisation method presented here is not part of the core work of this thesis, and is presented to demonstrate future potential for tools based on these classes. The algorithm used is similar to the *Marching Cubes* algorithm [LC87]. The future incorporation of a better algorithm is a trivial task as all the method calls and data structures for the polygonisation are already in place.

All objects of `FrepType` contain information about a bounding box for their solid material and a level of detail to which they should be rendered. (The level-of-detail data type is described in Section 7.2.1 of this chapter.) The space within the bounding box for the geometry is split into the number of voxels specified by the level of detail, each voxel having the same dimensions. The voxels form a three-dimensional grid contained within and exactly bounded by the bounding box. Any voxel corner vertex that does not lie on the face of the bounding box is a corner point for three other voxels. For the instance of `TopologyType` being considered for polygonisation, the initial phase of the algorithm is to sample the function representation of the geometry at each corner vertex of all the voxels calling the method `f()` with the coordinates of the vertices. The results of these samples are stored in a three-dimensional array data structure.

Each voxel is then considered in turn. Each voxel has eight vertices and each vertex can either be inside, outside or on the surface of the geometry represented by the instance of `FrepType`. The two cases for "inside" or "on the surface of" are combined into one making a total of 256 possible cases to consider for the way in which a geometric shape can be polygonised within a single voxel. Each of these 256 cases is considered separately in the second phase of the algorithm. Two special cases exists, where all the vertices lie within or on the surface of the shape, or all the vertices lie outside the shape, when the algorithm takes no action and produces no polygons. In all other 254 cases, at least one polygon is created.

Each edge of a voxel with end vertices $\mathbf{p}$ and $\mathbf{q}$ is split at point $\mathbf{s}$ if the edge crosses the surface boundary. This point $\mathbf{s}$ is a linear estimation of the actual position of the surface of the geometry, found by the solution of

$$tf(\mathbf{p}) + (1-t)f(\mathbf{q}) = 0 \tag{8.11}$$

with respect to t to define

$$\mathbf{s} = t\mathbf{p} + (1-t)\mathbf{q} \tag{8.12}$$

Each point $\mathbf{s}$ that can be found for an edge is used as the vertex of a triangle that contributes to the polygonisation of the whole geometry. If there are more than three connected edges of the voxel that can be split, there will be more than one triangle contributed to the overall polygonisation by this voxel. In the current implementation, an arbitrary choice is made without reference back to the geometry about how it is best to do this, which leads to a strange edge effect in the geometry such as warped shapes, as illustrated in Figures 8.8 and 8.7. An improved algorithm would make this decision with further sampling of the function representation[7].

Figure 8.10 shows a voxel with the eight vertices at which the function representation for the solid geometry is sampled. In this example, only one vertex is inside the geometry and this is depicted by a filled circle. Three edges that are cross a surface boundary for the solid are split to define the edge points of a triangle, shown in the figure by a dotted line. This triangle is used as part of the polygonisation of the solid geometric shape.

8.4.4 Rendering Issues

The *empirical world* server application calculates the polygonisation of geometric shapes described by function representations. In this section, statistics on the rendering times for some example geometric shapes are presented. These statistics:

[7]An appropriate way to improve the method would be to sample the function representation at a point in the centre of the voxel and split the problem of polygonising the geometry inside the voxel into polygonising inside six equal sized, square based pyramids.

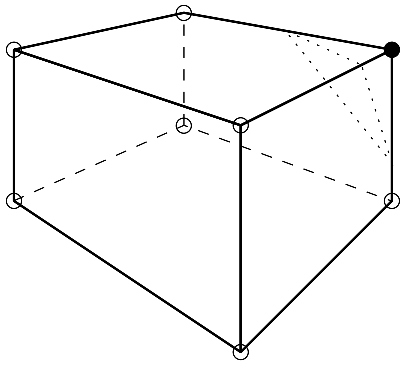


Figure 8.10: One example case for a voxel used in polygonising some solid geometry.

- demonstrate the genuine improvement over the scope for rendering in EDEN (cf CADNORT in Section 3.4.2);
- illustrate the potential for realistic real-time interaction;
- illustrate the network costs involved in a client/server application;
- demonstrate the potential use for Java as a programming platform for shape modelling.

The *empirical world* server produces statistics of the length of time each phase of the algorithm took to compute[8]. Some examples of these timings, averaged over five samples, are shown in units of seconds in Table 8.1. In this table, "*phase 1*" columns represent average timings for the sampling of the function representation at the corner points of the three-dimensional voxel grid. The "*phase 2*" columns are overall timings for the computation of polygons inside each voxel. The "*transmit*" columns show the time that it takes for the polygon image to be transmitted from the server to the client[9]. The polygonisation of three different solid shapes are timed, each at three detail levels of $10 \times 10 \times 10$ voxels, $20 \times 20 \times 20$ voxels and $30 \times 30 \times 30$

[8]Timings were recorded by executing the *empirical world* server on a Pentium 166Mhz processor with 32Mb memory, using Microsoft's *Windows 95* operating system and Sun Microsystems' *Java Virtual Machine*.

[9]This is dependent on the speed of the local network and included only for a guide to the wait between definitions being typed and displayed.

Shape	`detail 10 10 10`			`detail 20 20 20`			`detail 30 30 30`		
	phase 1	*phase 2*	*tran-smit*	*phase 1*	*phase 2*	*tran-smit*	*phase 1*	*phase 2*	*tran-smit*
Sphere	<0.05	0.07	0.26	0.08	0.66	1.29	0.28	3.09	3.01
Blend of a box and a cylinder	<0.05	0.06	0.27	0.14	0.50	1.10	0.50	2.25	2.43
Twist of previous blend	<0.05	0.06	0.34	0.19	0.48	1.51	0.64	2.00	3.62

Table 8.1: Average timings in seconds for polygonisation by the *empirical world* server.

voxels. The first row is for a sphere, the second for the blend of a box and a cylinder and the third for a twist of the blended shape. Timings marked “`<0.05`” were below accurate measurement thresholds (large standard deviation) but were consistently less that 0.05 seconds.

The table shows that phase 1 in these examples typically takes less time than phase 2 and that all timings increase with the level of rendering detail. As the complexity of the shape increases by the number of operators that define the shape (number of levels below the shape in its dependency structure), the time to compute the function representation increases for the same number of points, as shown in the “*phase 1*” columns. The sampling of a sphere takes fewer nested computations with only one level in its dependency structure than the sampling of a twisted blend of a box and a cylinder with a least three levels in its dependency structure. All timings along the rows are proportional, within a margin of timing error, to the number of points at which the function representation are calculated. For a detail of $10 \times 10 \times 10$ this is $11^3 = 1331$ points, $20 \times 20 \times 20$ this is $21^3 = 9261$ points, $m+1 \times n+1 \times r+1$ this is mnr points.

The use of *empirical world* scripts in such a way that they can be interactively explored requires that the user in control understands ways in which they can fine tune their model to optimise the quality and usefulness of modelling interaction.

The methods available in the open-ended client user environment for the user to enhance the shape modelling experience include:

- balancing quality of image with speed of rendering;
- selecting to view an entire geometric part or its subcomponents;
- restructuring the script to reduce the number of levels in its dependency structure.

The client tool is considered as an instrument for shape modelling.

8.5 Example of an Empirical World Script

In this section a case-study of the creation and interaction with some geometry in *empirical worlds* is presented. The geometry represents a curtain pole support. Figure 8.11 shows a sketch of the support in front and side views. This sketch can be considered as specifying the real-world object for the model. The figure is labelled with parametrisations that are considered to be important in the description of the geometry of the shape.

This case-study illustrates a shape that could easily be represented with computer software for CSG modelling[10]. Section 8.5.2 illustrates some redefinitions of the geometry of the support that would not be possible in many CSG modelling packages. The *CSG tree* common to many solid modelling programs is replaced in a script with a dependency structure. It is also interesting to note that parametric modellers such as Parametric Technology's *Pro/Engineer*[11], a constraint based system, requires the recalculation of a whole piece of geometry when one defining

[10] Matra Datavision's *Prelude Solids* package is an example of such a tool. See `http://www.matra-datavision.fr/Products/Family/Prelude/index.html`.

[11] See `http://www.ptc.com/products/mech/proe/index.htm`.

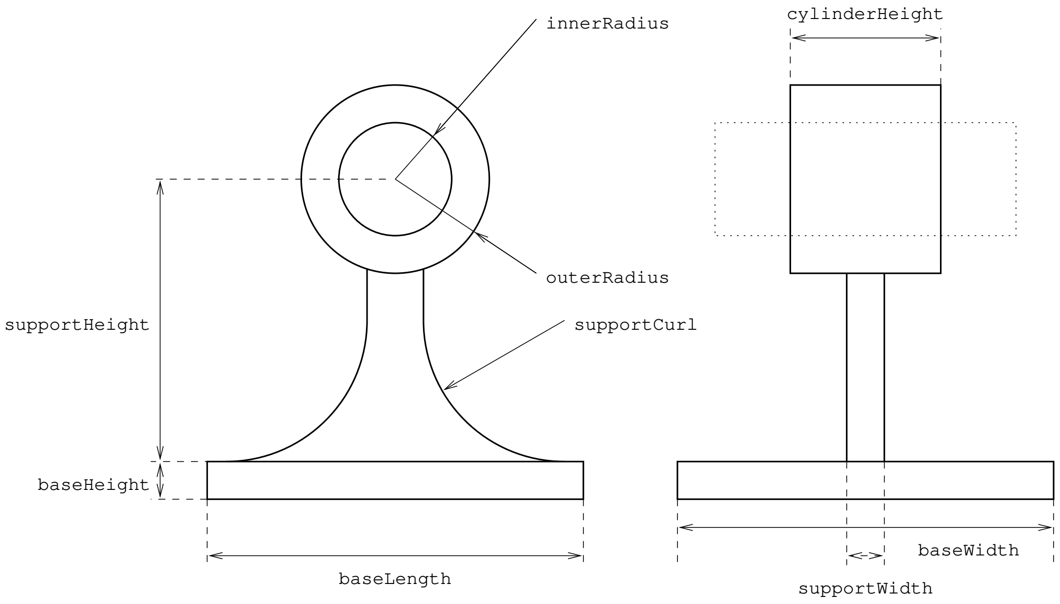


Figure 8.11: Two-dimensional representation of a curtain pole support with parametrisations.

parameter is altered (see Section 2.2.2). With the *empirical worlds* tool, only components of the overall geometry of the support that need updating are updated, not all the components.

8.5.1 Constructing the Curtain Pole Support Script

The parametrisations of Figure 8.11 are transferred into an *empirical world* script for the curtain pole support as a demonstration of a suitable process for the creation of a model. Firstly, a box is constructed to represent the base of the support. All primitive shapes are initially created centred at the origin and then translated to their final positions. (The implicit references `halfBaseLength` and `halfBaseHeight` are introduced here for use in definitions that appear further through the script.)

```
baseHeight     = 0.2
baseLength     = 2.0
baseWidth      = 2.0
halfBaseLength = divide(baseLength, 2)
halfBaseHeight = divide(baseHeight, -2)
baseOriginal   = box(baseLength, baseWidth, baseHeight)
```

The second stage of the modelling process is to create the cylinders that will form the head part of the curtain pole support. This head will be constructed from a solid cylinder with another cylinder used as a tool to remove material from its interior. The inner tool cylinder is set to be taller (longer) than the body cylinder, so that problems associated with rendering coincident faces when their are slight inaccuracies in floating point arithmetic are not introduced.

```
cylinderHeight   = 0.8
holeHeight       = multiply(cylinderHeight, 2)
outerRadius      = 0.5
innerRadius      = 0.3
outerCylOriginal = cylinder(cylinderHeight, outerRadius)
innerCylOriginal = cylinder(holeHeight, innerRadius)
```

The support that connects the base and the head of the curtain pole support is the most complex piece of geometry. It is constructed from a thin box with two cylinders and two boxes cut away from it. The original geometry for these components is described by the segment of script shown below. The original body of material for the support has identifier `supportBodyOriginal`, the cylinder that will be the tool that cuts away material to make the curved shape towards the base of the support is `cutCylOriginal` and the material cut away to make the neck of the support is `cutBoxOriginal`.

```
supportHeight       = 1.5
supportCurl         = 0.75
supportWidth        = 0.2
supportLength       = 0.3
doubleSupportCurl   = multiply(supportCurl, 2)
halfSupportLength   = divide(supportLength, 2)
halfSupportHeight   = divide(supportHeight, 2)
supportBodyOriginal = box(baseLength, supportWidth,
                                supportHeight)
cutCylOriginal      = cylinder(baseWidth, supportCurl)
cutBoxOriginal      = box(doubleSupportCurl, baseWidth,
                                supportHeight)
```

The third stage is to translate the `cutCylOriginal` and `cutBoxOriginal` into their correct locations to become tools to cut the support. Their respective translated geometries are identified as `cutCyl` and `cutBox`.

```
cylCenterX = add(halfSupportLength, supportCurl)
cutCyl     = translate(cylCenterX, 0.0, supportCurl,
                                cutCylOriginal)

boxCenterX = add(supportCurl, halfSupportLength)
boxCenterZ = add(halfSupportHeight, supportCurl)
cutBox     = translate(boxCenterX, 0.0, boxCenterZ,
                                cutBoxOriginal)
```

At this point, the tool for cutting the support geometry can be created as a whole without creating separate cylinder and box tools for the left and right hand sides of the cut. The union of `cutBox` and `cutCyl` is created and then rotated around the z-axis by π radians. The union of the original tool and the rotated copy form the definitions of the complete tool `supportTool`.

```
detail1 = detail 4 4 4
detail3 = detail 30 30 30
supportTool1    = blendUnion(detail1, 0.0, 1.0, 1.0,
                                cutBox, cutCyl)
supportTool2    = rotate(rotation 0 0 1  1pi, supportTool1)
supportTool     = blendUnion(detail1, 0.0, 1.0, 1.0,
                         supportTool1, supportTool2)
```

It is also necessary to translate each piece of component geometry of the final shape to its final position ready for the application of point set combinations. The base will be called `base`, body material of the central neck support is called `supportBody`, the top cylindrical shapes are called `outerCyl` for the body material and `innerCyl` for the tool to cut the hole.

```
base         = translate(0.0, 0.0, halfBaseHeight, baseOriginal)
supportBody  = translate(0.0, 0.0, halfSupportHeight,
                                   supportBodyOriginal)
outerCyl     = translate(0.0, 0.0, supportHeight,
                                   outerCylOriginal)
innerCyl     = translate(0.0, 0.0, supportHeight,
                                   innerCylOriginal)
```

The final stage of the modelling process is to combine the point sets that are now in their correct locations to form the final geometry of the curtain pole support. The central neck `support` is created by cutting the `supportTool` away from the `supportBody`. Then the support and the base are blended together to make the lower part (`lowerPart`) of the final shape. This is then blended with the outer cylinder of the head of the shape (`noHole`) before the material from the inner cylinder is removed to make the hole in the top of the object. The final geometry of the support is called `final`.

```
lowerPart = blendUnion(detail1, 0.05, 1.0, 1.0,
                                   base, supportBody)
noHole = blendUnion(detail1, 0.03, 1.0, 1.0,
                                   lowerPart, outerCyl)
final = cut(detail3, noHole, innerCyl)
```

In addition to the geometry of the shapes, it may be necessary to attach attributes of colour and texture to the geometry and place this geometry in a VRML world where it can be interactively explored. Figure 8.12 shows the curtain pole support script rendered in a VRML browser with a wooden texture. Relevant definitions to apply this image texture to the geometry are given below.

Figure 8.12: Image of the example curtain pole support (with no modifications).

```
wood      = ImageTexture { url ".../wood_floor.jpg" }
appear    = appearance( Material { } , wood)
tmp       = attribute(appear, final)
finalList = list(final)
w         = world(finalList)
```

8.5.2 Redefinitions of the Curtain Pole Support Script

The curtain pole script can be considered as a template script for a whole family of geometry with similarities to the original support. The possible versions of the geometry can be interactively explored in *empirical worlds* by making redefinitions of any identifier of a script. Three examples of redefinitions of the curtain pole support illustrated in Figure 8.12 are presented in this section.

Figure 8.13: Image of the curtain pole support with an extended neck.

The conversion of the diagram of the support in Figure 8.11 into a script of definitions required the identification of some defining parameters of the three-dimensional geometry. One of these is the `supportHeight` that defines the height of the *neck* of the support that connects the cylindrical head to the box base. In a real-world location, there may be a scenario where the shape of a window required the curtains need to hang further away from the wall than with the standard part. Suitable geometry can be achieved by one simple redefinition of the *empirical world* script for the pole.

```
supportHeight = 2.0
```

The height of the pole is increased from `1.5` in the original script to `2.0` by this redefinition. The new state of the geometry is rendered in Figure 8.13.

Another way to make a new version of the support is to use it with an operation such as tapering or bending. The redefinitions shown below cause the whole geometry of the support (`final`) to be linearly tapered along the z-axis (`finalTapered`) in such a way that the cylindrical head is stretched out of shape. The redefinitions to achieve this are shown below and an image of the redefined geometry is illustrated in Figure 8.14 with a marble-like image texture applied.

```
final = cut(detail1, noHole, innerCyl)
finalTapered = taperXZ(detail3, -0.35, final)
finalList = list(finalTapered)
```

In the first of these three redefinitions, the level of detail of `final` is reduced. The reason for this is that `final` is no longer the focus of the modelling process, therefore reducing the detail level will speed up the overall response time for the system. The `finalTapered` redefinition is an implicit definition representing the tapered geometry. In the final redefinition, the tapered geometry is selected to be viewed in the VRML browser by setting it to be the only element of `finalList`. Notice how both the original geometry and the tapered geometry now exist as their own separate entities within the same script simultaneously, available for future modification or as arguments to other implicit redefinitions.

The final example of redefinition in this case-study examines the modification and replacement of a component of the geometry. In the redefinitions below, a new version of the support part is created (called `supportTwist`) that is a twisted version of the original `support` geometry along the z-axis. This new version of the support geometry replaces the original in the definition of the lower part of the geometry `lowerPart`. A close up image of the support to cylindrical head join is shown in Figure 8.15. The redefinitions associated with the twist are shown below.

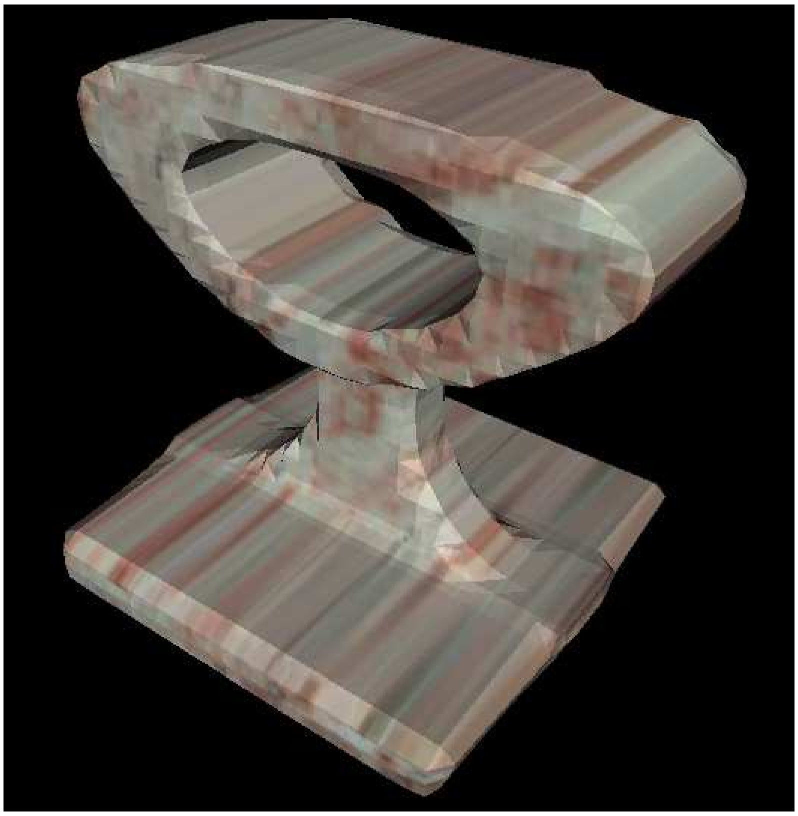


Figure 8.14: Image of the curtain pole support tapered along the z-axis.

```
final = cut(detail3, noHole, innerCyl)
supportTwist = twistZ(detail1, 0.78539, supportBody)
lowerPart = blendUnion(detail1, 0.05, 1.0, 1.0,
                              base, supportTwist)
finalList = list(final)
```

Having created the definition of `supportTwist`, it is easy to subsequently return to the original support geometry by substituting `support` for `supportTwist` in the definition of `lowerPart`.

These case-studies have shown how one piece of geometry is represented is represented in an *empirical world* script, a user can interactively modify and explore that geometry in an open-ended way. What has not been demonstrated is the ability to then use this geometry or its component parts to create other geometry. The support geometry and any of its component parts can be used in any operation (CSG, blending, morphing, bending) to build or model other geometry. In one snapshot of state, the script of definitions is a specification for shape that can be stored in a plain text file and communicated to other people by any means of textual transmission, such as e-mail, letter or computer chat program[12]. The script of definitions for the curtain pole support represents a computer-based artefact for the exploration of the geometry for a particular shape.

[12]The *Arithmetic Chat* application, presented in Section 6.5, for the interactive discussion of scripts can be used as a template for the implementation of a geometric chat application.

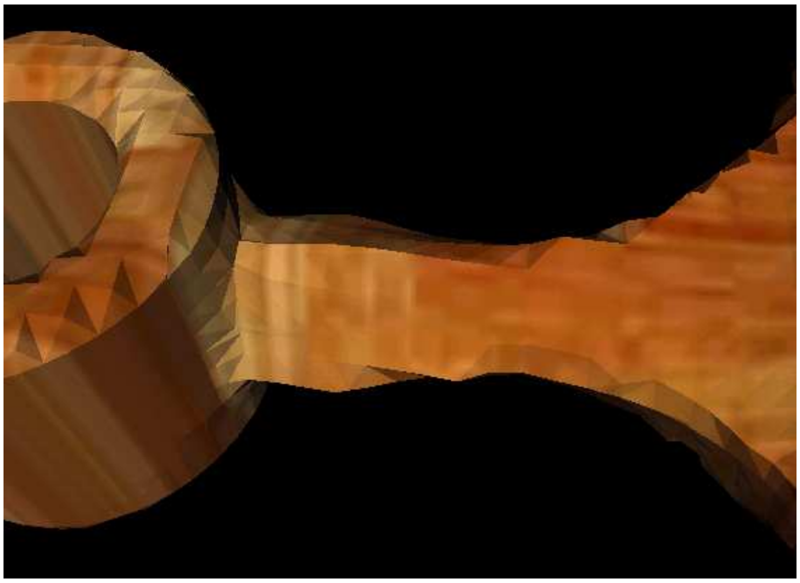


Figure 8.15: Image of the curtain pole support with a twisted neck.

Chapter 9

Conclusions and Further Work

9.1 Introduction

In this final chapter, the research work presented in Chapters 4 through to 8 of this thesis is reviewed with reference to:

- the research aims of the work as set out in Chapter 1;
- the requirements for tools identified in Chapter 2;
- the technical issues in Chapter 3.

This is followed by some suggestions for further work (Section 9.3) and a proposal for a new application, based on spreadsheet ideas, that can support interactive open development of geometric models as well as other models requiring more complex, general data types than are supported by existing spreadsheet applications.

In Chapter 1, the roles of programmer, user and systems expert are discussed. Empirical modelling supports the conflation of these roles. In the chapters that follow (particularly 3 through to 6) the emphasis is on abstract models, issues of data representation and programming toolkits. Support for high-level definitive scripts (limited in EDEN — see Section 1.2.2) can be better achieved by discarding

EDEN in favour of a new generation of tools that can be best implemented using the DAM Machine or the JaM Machine API. EDEN is reasonably successful in conflating the roles of user and programmer, but only the next generation of tools can enable a programmer to construct new instruments for empirical modelling (see Figure 1.1). These new tools will also provide improved interactivity, and data representations that support a wider range of applications — including solid shape modelling — than is possible at present.

One of the aims for this work (see Section 1.2.2) has been to address issues not tackled by EDEN. The DAM Machine and the JaM Machine API are primarily concerned with the dependency maintenance aspect of EDEN, and the management of data becomes the task for a programmer. The DAM and JaM toolkits address:

- aspects of representing the privileges of interacting agents using definition permissions (JaM Machine API);
- the provision of greater flexibility to represent a broader range of data types (empirical worlds);
- efficiency by using compiled code for operators (DoNaLD to DAM translator) and improving dependency update algorithms (block redefinition algorithm);
- portability through the use of the Java programming language (JaM Machine API);
- the dynamic creation/elimination of definitions (management of the definitive store of the DAM Machine);
- generalised representation of data types (JaM Machine API and patterns of bits for DAM words).

9.2 Review of Definitive Programming

In Section 3.5 the transformation of all representable types of data to serialised data types in a definitive notation is proposed as a solution to many of the technical issues. The JaM Machine API allows a programmer to create special atomic data types for definitive notations by extending the class `JaM.DefnType`. The *empirical worlds* case-study in Chapters 7 and 8 illustrates that this is an effective method for modelling solid geometry using definitive notations. This is a new approach to definitive programming that combines:

- dependency relationships between observables;
- data structure relationships and constructors;
- dependency through expression evaluation;
- dependency between sets (such as point sets)

in unified dependency structures.

Figure 9.1 illustrates such a combined dependency structure. The script shown represents a geometric model of a hammer with a box for a head and a cylinder for a handle[1]. The length, width and height of the head are the defining parameters from which all other parameters of the model are calculated. The figure also shows a description of each parameter and the dependency structure associated with the script.

In this example, the parameters can be classified as follows:

Scalar Values (`a1` ... `a7`) Numerical values that represent defining parameters for other parts of the model. The dependency between `a6` and `a1` is given by a mathematical expression of the value of `a6` (with automatically generated parameter `a6_2`).

[1]There is a diagrammatic representation of the hammer in figure 9.2.

`a1 = 5`	`a1` is hammer head length
`a2 = 3`	`a2` is hammer head width
`a3 = 2`	`a3` is hammer head height
`a4 = divide(a3,2.5)`	`a4` is hammer handle radius
`a5 = multiply(a1,2)`	`a5` is hammer handle length
`a6 = multiply(a1,-1)`	`a6` is handle shift y amount
`a7 = divide(a1,10)`	`a7` is handle shift x amount
`b1 = makePoint(a7,a6,0)`	`b1` is handle translation vector
`c1 = box(a1,a2,a3)`	`c1` is hammer head, no hole
`c2 = cylinder(a4,a5)`	`c2` is hammer handle original
`c3 = translate(b1,c2)`	`c3` is hammer handle translated
`d1 = union(c1,c3)`	`d1` is complete hammer
`d2 = cut(c1,c3)`	`d2` is hammer head with hole

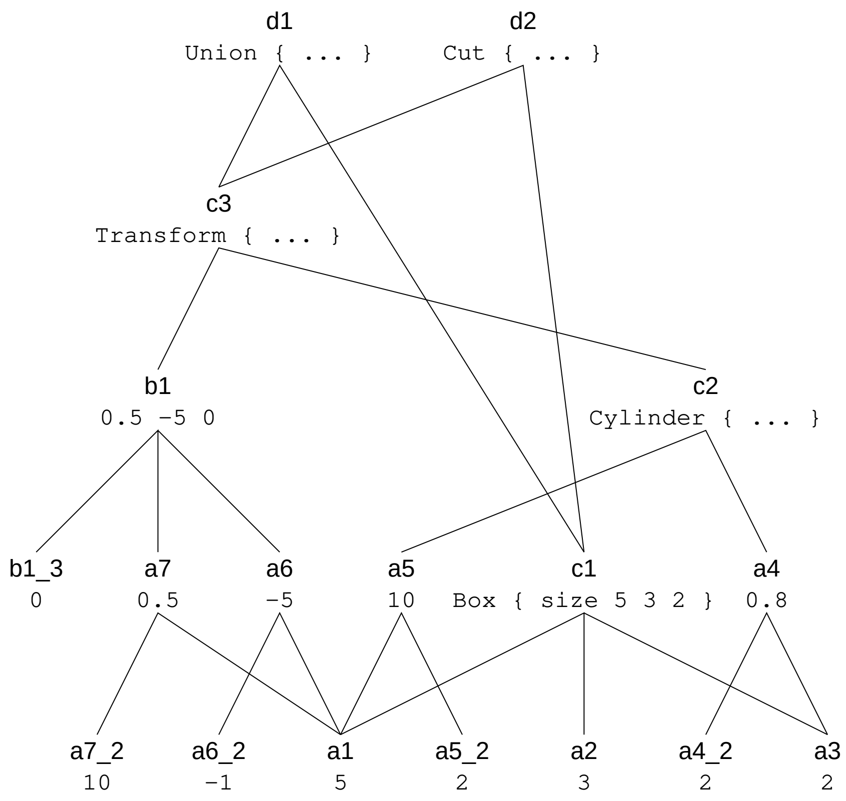


Figure 9.1: Script and dependency structure for a hammer model.

Data Structures (`b1`) A value for a three-dimensional point is dependent on its three sub-component parts. The dependency between `a6`, `a7` and `b1` is for the creation of an instance of a data structure by the "`makePoint`" operator.

Primitive Shapes (`c1`, `c2`) Point sets with their parameters forming values in internal data structures. The dependency between `c2`, `a4` and `a5` defines indicates that the scalar values are parameters in the construction of the point sets for the cylinder primitive shape that represents the hammer's handle.

Point Sets (`d1`, `d2`) Point sets constructed from boolean operations on other point sets. The dependency between `c1`, `c3` and `d1` constructs `d1` as the union of the primitive shape point sets.

These four categories for types of dependency can also be interpreted as different forms of agency: there are agents that are responsible for dependencies that form data structure, other agents responsible for expressions with their evaluation (possibly represented by several definitions to simulate an expression tree), and so on. This categorisation does not include all possible combinations of types of dependency. For example, the translated cylinder shape `c3` is actually a combination of an existing point set with some parameters that represent a translation. This defines a new point set. In Sections 9.2.1 through to 9.2.5, the relevance to empirical modelling of the integration of different types of dependency and data structure is discussed.

9.2.1 Empirical Modelling with Geometry

Cognitive Artefacts for Geometric Modelling

The *empirical world* tool demonstrates the construction of geometric cognitive artefacts (e.g. the curtain pole support of Section 8.5). By experimenting with the parameters defining the model, replacing component parts of the model or using the constructed model as part of other models, the script of definitions describing the geometry is available for open development and exploration of shape. Shape modelling is not supported in the *Tkeden* tool. For tasks other than shape modelling, it would also be possible to build new applications that use the *empirical world* classes as a rendering method for providing three-dimensional, solid shape metaphors for cognitive artefacts. With appropriate operators, it is possible to represent the dependency between observables for the artefact model and its graphical representation as shape with definitions.

Animating Geometric Models

With the *empirical world* tool, it would not be possible to animate the timepiece artefacts of Section 2.3.2 with three-dimensional geometry in real time. There may be a delay of more than one second in the time taken to render the model, both in terms of polygonisation and transmission of the model to a VRML browser. This is because the tool is optimised for exploring implicitly defined shapes by flying around them when they appear in a browser. This is an implementation-specific problem and when programming libraries become available for Java to support three-dimensional modelling[2], it will be possible to control computer graphics hardware more directly from JaM classes. In addition, it may be possible to apply techniques similar to those of the DoNaLD to DAM Machine translator of Section 5.4 to animate smoother rotation of the second-hand of a clock than is possible in *Tkeden*.

[2]The Java3D API is under development and currently exists in an alpha-test version. Initial tests by the Manufacturing Group at the University of Warwick have shown this to be a very powerful toolkit for implementing fast rendering of three-dimensional shape.

Support for Collaborative Modelling

By using classes of Java programming APIs that support threading processes and network communications, it is possible to support simultaneous and incremental development of cognitive artefacts by several cooperating agents. The support for this in the JaM Machine API is implemented through setting permissions for each definition and providing security by a username and password system. Each human agent can interact with a model at a workstation through a graphical interface, such as a web browser, that can display geometry that is a visual metaphor for their view of an artefact. It is also possible to take advantage of the threading of processes to introduce automatic agents into models, with guarded protocols to take action. It is the job of a programmer to build interfaces using the JaM Machine API and manage the scripts of definitions — the advantage of this method is that a script that represents an artefact and its current state can be distributed between human agents for interactive collaborative working[3].

9.2.2 Empirical Modelling for Geometry

Shape Modelling

The general data representation technique provided by the JaM Machine API has been used to provide an extendable definitive notation for geometric modelling. The *empirical world* class library supports all the geometric shape primitives available in the VRML notation and builds on these through the use of the function representation for shape with a polygonisation algorithm for their display. The basic standard primitives that are used in CSG modelling are combined in *empirical worlds* with skeletal-based implicit representation for shapes. Other shape representations, in-

[3]It is interesting to note that a lot of tools that support collaborative working are rather static, working on publish/subscribe models where one agent works on a model, publishes it and then another user subscribes and works on the model. The JaM Machine API has the potential to support simultaneous and interactive collaborative working.

cluding surface models, can be added to *empirical worlds* by introducing a class to represent their function representation together with a VRML-like syntax for describing their parameters. These could include, for example, the Bézier volume function representations described by [MPS96]. All the operations on the point sets are closed — they produce other point sets in function representation. For example, it is possible to cut a shape constructed by CSG techniques away from a shape created by using the BCSO representation.

Dependency and Constraints

Relationships between geometric entities can be expressed as definitions in scripts. The parameters and structure of models can be efficiently updated given a change to a value by the block redefinition algorithm. There is no need for a constraint satisfaction algorithm and a script of definitions will never be over- or under-constrained[4]. The classes support the open development of models and enable a user to incrementally construct models from their experience. The models themselves then generate new experience. In the hammer model script shown in Figure 9.1, the box and cylinder could be constructed initially and then subsequently be located and scaled. Alternatively, the order of construction could reflect the order of the script. The proportions in the model having been established, the hammer head shape can be replaced by a more realistic shape.

CADNO and Empirical Worlds

Empirical world scripts can potentially be used as templates in the realisation section (see page 50) in a new implementation of the CADNO notation. The alternative to this is to implement data types for representing topological structures (complexes) as part of the *empirical world* tool. In practice, it is possible to define points to

[4]See the discussion of two way constraints in Section 9.2.3.

represent abstract observables in *empirical worlds*. For the table model that is used as a case-study for CADNO in Section 2.4.3, it is possible to introduce definitions of the parameters table height, length, width and so on into *empirical world* scripts. In this way, the abstract corner points of the table — that do not physically exist — are incorporated as part of the geometry. One of the features of CADNO is the possibility to restrict the parameters of certain ranges. To do this with *empirical world* classes, a programmer needs to implement methods that prevent definitions for values outside specified ranges, either by intercepting definitions before a call to `JaM.Script.addToQ` or by starting threads to continuously police the ranges of certain values.

9.2.3 Technical Challenges Reviewed

Different Kinds of Copy

In Section 3.2.1 of the thesis, the problem of copying assemblies of definitions is considered. Since no definition represented by the DAM Machine or in a `JaM.Script` defines a value that is an assembly[5] and since no function composition is allowed, a copy of an observable's value is exactly a mirror copy, photographic copy or reconstruction (with reuse) copy. A programmer using the JaM or DAM Machine programming toolkits can choose which kind of copy they wish to make by using various methods, without concerns for how to copy any subcomponents of assemblies of definitions. These methods are described in Table 9.1.

Higher-order Dependency

In the absence of assemblies of definitions, the higher-order dependencies described in Section 3.2.2 do not pose a significant technical challenge.

[5]In a DAM representation, every observable is represented by a pattern of bits. In an instance of a class that extends `JaM.DefnType`, every value has a string representation that can be printed and recognised.

Type of copy	**DAM Machine**	**JaM Machine API**
Photographic	Copy the value bit by bit into another memory location in definitive store. No dependency.	Call `JaM.Script.printVal` method for the original definition, change the identifier to the name of the copy. Use this string to call `JaM.Script.addToQ`.
Mirror	Create a subroutine that looks up the value stored at the address passed to it and returns this value. Use this subroutine for the operator associated with the copy. The argument to the operator should be the address of the original value.	Use the `JaM.JaMID` operator (uses `printVal` method to maintain dependency) to create an implicit definition where the the argument is the original and the identifier is the copy.
Reconstruction	Copy the associated operator from the original to that associated with the copy. Also, construct copies of the dependencies list for the original and associate this with the copy.	Call `JaM.Script.printDef` method for the original definition, change the identifier name to the name of the copy. Use this string to call `JaM.Script.addToQ`.

Table 9.1: Methods for copying in DAM and JaM.

Such dependencies can be handled by introducing serialised data types that can represent all observables values at level 0 and operators to represent dependency between these values at level 1. As it is possible to conceive a data type for the representation of a definitive script (a JaM class that extends `JaM.DefnType` and contains a field for an instance of a `JaM.Script` class), dependencies at higher levels representing patterns of dependency between scripts can be considered at level 0 and 1 also. This scripts-within-scripts model is unlikely to provide the detailed support that tools designed to support higher-order dependency will be able to provide[6].

[6]Gehring is currently developing a programming API implemented in Java that supports empirical modelling with higher order dependency, entitled *MODD*.

Although most observed dependency in the world can be described by definitions where one observable is dependent on another, for some dependencies this distinction is unclear. For example, geometric constraints in parametric modelling packages can constrain two lines to be parallel to one another. It is desirable for a tool that supports empirical modelling to be able to handle constraints. One way to deal with such constraints between observables using the JaM Machine API is to choose one direction for the dependency and then introduce an automatic agent (maybe with a thread) who observes all definitions that are being placed onto the queue of redefinitions. If observable a depends on the value of observable b and the value of a is redefined, then the direction of the dependency is switched by the automatic agent. The value of b is redefined to depend on the value of a. There needs to be a mechanism provided by a programmer of an application to allow a user to be able to create and destroy these automatic agents.

Moding and Data Structure

The issue of moding variables in a definitive script, introduced in Section 3.3.1, is greatly simplified by considering all data as representable by special atomic types. The level at which a variable is defined implicitly or explicitly can be established by looking through a definitive script. In the JaM Machine API, data structure is expressed via the fields of classes that extend `JaM.DefnType`. These classes are instantiated to represent values for observables in definitive scripts. This underlying data structure is separated from dependency maintenance by only accessing and setting the values of the fields of an instance through method calls and carefully defined operators, which are classes that extends `JaM.DefnFunc`.

9.2.4 The Contribution of the DM Model

The DM Model described in Section 4.2 can be used to reason about dependency structures without any need to consider application-specific information. It is possible to build a DM Model of a script of definitions and to analyse the dependency structure in that script. A dependency structure diagram is a visual formalism for a definitive script. It can be used to examine how redefinitions cause the update of values to propagate through different dependency structures. Consider, for example, counting compare/exchange operations in the sorting algorithm case-study in Section 4.4.3. Through this analysis, it is possible to determine ways to improve the stimulus-response time of interaction with models by reducing the upper bound for the number of updates.

The DM Model can be used as a basis for discussing and optimising update strategies. The block redefinition algorithm is one possible way to keep all values up-to-date given a block of redefinitions. Other update strategies include introducing threads for each vertex that regularly check to see if a value is up to date, with the potential to take advantage of parallel processing hardware. Another method is lazy evaluation, where every time a vertex x is referenced (its current value is requested), a tool checks to see if any recent redefinitions have effected its current value. All the dependencies of the vertex are updated prior the update of s. The DM Model can be used as a basis for comparing and contrasting different update strategies. Consideration of this model lead to the development of the block redefinition algorithm, which out performs existing update algorithms when more than one definition is redefined simultaneously.

The case-studies for finding minimum and maximum values and sorting using dependency maintenance illustrate how a dependency structure can be used as a form of algorithm. Empirical modelling encourages a user to program a computer without them realising that that is what they are doing. Incrementally constructing dependency structures, as in definitive programming, does not require a user to reason in the computation-oriented way that a programmer needs to when implementing an algorithm to perform a task. The process only requires the identification of patterns of dependency that reflect personal insight into the real world. This process supports the conflation of the roles of user and programmer.

9.2.5 Combining Definitive and Object-oriented Methods

In implementing new definitive notations via the JaM Machine API, object-oriented programming techniques are used to represent data types and operators of an underlying algebra for a programmer to implement new definitive notations. In general, this approach to implementing new notations is successful where all classes used by the notation are written specifically for use with JaM. Problems can occur when a programmer tries to integrate classes from other programming APIs with classes that extend `JaM.DefnType`. Classes in standard APIs generally have their own private data fields that can only be accessed by method calls. This means that there are aspects of the current state of an instance of such a class that are hidden from the implementor of the JaM class.

For example, the `java.awt.TextArea` class [CH96] can be instantiated to create a graphical component of a user interface that can be used for displaying text and enables a user to enter and edit text[7]. The current state of the text can be set and read at any time by calling methods of an instance of this class but there is no way to detect the text changing as a user types each character. The object structure hides the key presses from other classes and places the character onto the screen *automatically*. If there is an observed dependency between a certain pattern

[7]The input and output components of the *empirical world* client are text areas, as shown in Figure 8.9.

of characters and another observable, such as a child types a rude word and a teacher tells them off, the use of the standard `TextArea` component is inappropriate[8].

The object-oriented approach to programming requires a programmer to form abstract classifications of real world entities by choosing data fields to represent their internal state. It is then necessary to prescribe the methods for communication with instances of a class. With the JaM Machine API, it is still necessary for a programmer to classify real world entities but there is no need to commit to the methods by which classes communicate. Where the communication between objects serves to maintain indivisible relationships between observables, dependency relationships can be directly introduces using the JaM Machine API. This process requires a library of objects that represents the operators (classes that extend `JaM.DefnFunc`) to be used for this communication. The advantage of this technique is that it is possible for a user or programmer to experiment with the communication between objects on-the-fly through redefinitions. No compilation is required during this process.

9.3 Further Work

This thesis introduces new tools to support the development of new definitive notations and other utilities for empirical modelling. The purpose of the case-studies presented is to demonstrate "proof-of-concept" for these tools. There is the potential to:

- develop new case-studies and applications with these tools;
- improve existing case-studies such as *empirical worlds*, leading to more comprehensive JaM Notations and associated applications;

[8] Many graphical user interfaces use constraint solving methods to locate graphical components inside windows. Definitive programming would seem to be a good way to represent the dependencies between components, as demonstrated by SCOUT [Dep92]. To achieve this successfully using the Java API packages would require rewriting a lot of the classes of the API so that they extend `Java.DefnType`.

- improve on the underlying dependency maintenance mechanisms and data representations in existing tools.

In Sections 9.3.1 through to Section 9.3.4 some suggestions are made for further work based on the research presented in the thesis.

9.3.1 The Future of the DAM Machine

The DAM Machine implementation is intended to be a proof-of-concept tool. The DoNaLD to DAM compiler demonstrates how it can be used as the dependency maintainer that supports a definitive notation. It would be interesting to see if it is possible to implement new versions of EDEN, SCOUT or ARCA on the machine with the potential to support more efficient animation. There may be other new definitive notations that are well suited to implementation supported by DAM. Further investigation is needed into using the machine for maintaining dependency between groups of low-level words in definitive store that represent high-level values.

There is also the potential to investigate new software and possibly hardware implementations of the DAM Machine software. The latest ARM processor (the "*StrongARM*") has a clock speed that is more than five times faster than the one used for the work in this thesis and an on-board processor cache that is large enough to accommodate the DAM Machine assembly code[9]. There is also the possibility of investigating a new version of the DAM Machine that is implemented over the Java Virtual Machine [MD97] and is platform independent, although such a tool may animate models significantly slower than direct implementations that use native assembly code instructions.

[9]Some minor alterations may be required to use the StrongARM processor. It uses the same instruction set, but due to pipelining optimisations and on-board chip cache memory the effect of branch instructions can be different.

9.3.2 Support for Collaborative Working

Support for concurrent development and use of models is integrated into the JaM Machine API. This has been demonstrated so far in an elementary case-study in Section 6.5. Further case-studies are required to investigate the potential for the distributed use of this feature by both human and automated agents. For example, it should be possible to distribute the *empirical world* server to support concurrent engineering.

Some improvements should be considered for collaborative working by the JaM Machine API that should be considered. The current implementation of JaM only allows one agent owner to be associated with a definition. This can be extended to support both agent owner and group ownership properties for definitions in the same way that files are protected on a filing system. It should also be possible to include a time and date stamp for each definition to support better management of scripts of definitions. There is also an existing problem with the JaM class libraries in that the code in them breaks Java applet security rules [Har97]. It is not yet possible to run a stand-alone version of a JaM-based application in a web browser client without a server. This will require the implementation of a new cut-down and secure version of JaM.

9.3.3 New Shape Representations in Empirical Worlds

Two shape representation techniques have been demonstrated in the *empirical world* class library: CSG-style modelling and BCSO modelling. It is possible to extend the representation of shapes to include surface modelling with Bézier surfaces, B-spline surfaces, NURBS [WND97] and so on. *Empirical worlds* would benefit from support for the representation of two-dimensional shapes, that can be extruded to form shapes in three dimensions. For example, consider a shape formed by embedding a solid (filled) circle and a solid square, each in two separated and parallel planes in three dimensions. A three-dimensional shape between the two planes can be formed from the morph between the two profiles in two dimensions. Further investigation is required to compare variational and parametric modelling techniques with empirical modelling for geometry.

CAD packages typically address far more than just geometric shape modelling. There needs to be support in the *empirical world* classes for data exchange from and into other applications, through the use of standards such as STEP [Org98]. It would also be interesting to investigate the generation of CAM machine tool code to make modelled shapes through the expression of dependencies between the geometric shape and the possible motions of the automatic tool. Another common use of CAD models is to generate *finite element analysis* (FEM) models to simulate and analyse the effect of stresses applied to different pieces of geometry.

The rendering algorithm currently used in the *empirical world* tools produces rough images at the edges of a geometric model. Future implementations should integrate new algorithms for rendering implicit shapes, both through polygonisation and ray tracing. In this way, it should be possible to support the presentation of models that include fine features in geometry, such as the hair growing on implicit shape representations by Sourin et al [SPS96]. Although it is possible to increase the detail level for rendering a shape using *empirical worlds*, this is currently limited by the available memory in the *empirical world* client. This is due to the fact that the current implementation creates a string that is parsed by the VRML browser to render a shape. It is possible to communicate more directly with the VRML browser through instances of Java classes specifically designed to support the VRML external authoring interface, and future implementations should take advantage of these features.

9.3.4 Spreadsheets for Geometric Modelling

There are many similarities between definitive programming and the use of a spreadsheet. Rather than identifiers in a script of definitions, a spreadsheet provides an identity to a value by its location in a grid of cells (*A1*, *C3* etc.). Every cell has a value and, if it is dependent on the values in other cells, a formula defining its value. The research in this thesis relating to the representation of data structures and dependency is also relevant to spreadsheet applications. Spreadsheet applications are contrived to support financial and mathematical models and the data types represented in cells are numbers, strings of characters or dates and times. It is possible to link spreadsheet values to defining parameters in other applications, such as AutoCAD, using *object link embedded* (OLE) mechanisms that are provided in some operating systems. The proposal in this section is that it is possible to use the JaM Machine API to expand the range of data types supported by a spreadsheet application.

Consider the *empirical world* script for a hammer model shown in Figure 9.1. The identifiers have been chosen to correspond to references for cells in a geometric spreadsheet and an example of such a spreadsheet is shown in Figure 9.2. The cells of this spreadsheet are larger than those of a standard spreadsheet and each cell contains three sections. The top of the cell is its defining formula (definition), the centre is a representation of the value currently stored in the cell and the bottom section contains a list of references provided to the value in the cell. The box shape in cell **C1** depends on the values in cells **A1**, **A2** and **A3**. The list of references shown correspond to parameters that might be useful for a user to make reference to in the definition of other cells. For the box, this includes a bounding box that contains all the corner points for the box and a centre point.

The complete hammer shape is represented in cell **D1**. The image of the

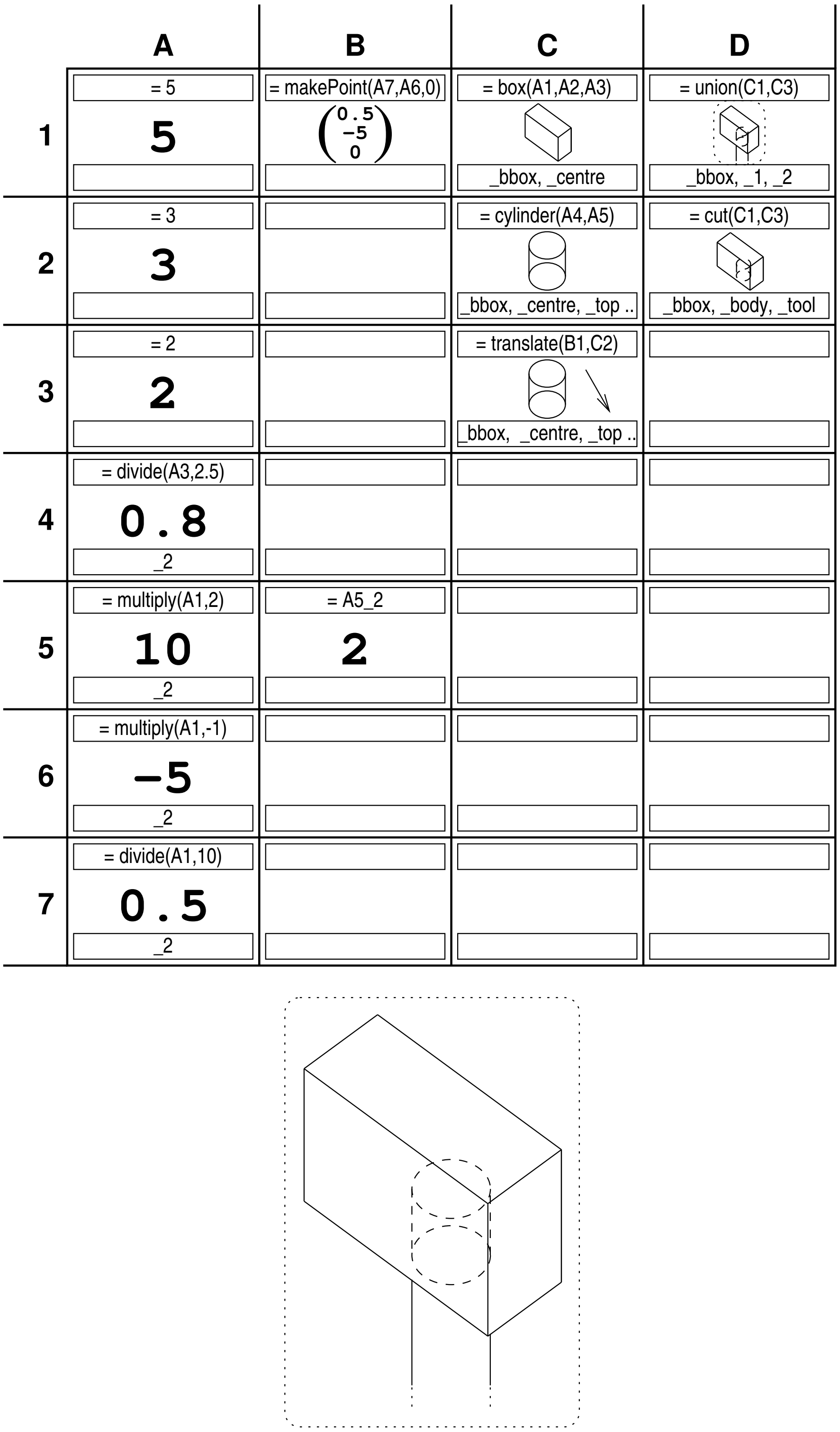


Figure 9.2: Geometric spreadsheet concept.

point set is only a *thumbnail* image. Ideally there should be a way to click on a cell with the result that a new window opens with a large rendered version of the geometric shape is displayed, as illustrated in the box at the bottom of the Figure 9.2. It should be possible to explore this shape in the same way that it is possible to explore a shape in a VRML browser[10]. This larger window can include graphical representations of the references to the geometry that can be selected by using a pointer.

A spreadsheet for geometric design, or any other application for which it is possible to write some JaM classes, allows a designer/modeller to integrate their work with other relevant data, such as financial data. It is possible to use a spreadsheet as a cognitive artefact for the explanation of a new design. A manager can experiment with parameters of a design indicated by a designer. They can describe a whole range of designs for assessment in one spreadsheet, rather than one design that will almost certainly require further modifications. The designer/modeller benefits from a tool that allows them to try out several *what-if?* experiments in the same way that a spreadsheet is often used as a way to test the financial feasibility for a project before its commencement.

In this thesis, the benefits of empirical modelling with geometry and empirical modelling for geometry have been discussed and explored. New tools to support general empirical modelling have been introduced and it shown that these tools can support three-dimensional shape modelling. The geometric spreadsheet is an obvious next step. It has the potential to support a large range of shape representations in one integrated application that can support complex data structures and interactive collaborative working on geometric models.

[10]A programming library like the *Java3D* API will be able to support this kind of interaction.

Appendix A

Appendices

Most of the material in the appendices to this thesis resembles programming manual documentation. The reader is referred to the world-wide-web addresses given below for the appropriate material.

A.1 Example DoNaLD Files

`http://www.dcs.warwick.ac.uk/~carters/doc/donald.html`

A.2 Exceptions in the JaM Machine API

`http://www.dcs.warwick.ac.uk/~carters/doc/jamexcept.html`

A.3 Arithmetic Chat Server Application

`http://www.dcs.warwick.ac.uk/~carters/doc/arithserv.html`

A.4 Basic Types and Operators in Empirical Worlds

`http://www.dcs.warwick.ac.uk/~carters/doc/basictyp.html`

A.5 Graphs in Empirical Worlds

http://www.dcs.warwick.ac.uk/~carters/doc/graphs.html

Bibliography

[ABCY98] J. Allderidge, W. M. Beynon, R. I. Cartwright, and Y. P. Yung. Enabling texhnologies for empirical modelling in graphics. In *Eurographics UK Conference Proceedings 1998, University of Leeds*, pages 199–214, 1998.

[ABH86] D. Angier, T. Bissell, and S. Hunt. DoNaLD: a line drawing notation based on definitive principles. Research Report 86, Department of Computer Science, University of Warwick, 1986.

[Adz94] V. D. Adzhiev. Agent–oriented aproach to exploratory geometric modeling based on function representation. In *Proceedings International Workshop on Shape Modelling: Parallelism, Interactivity and Applications*, pages 64–68. University of Aizu, Japan, September 1994.

[AF88] S. Ansaldi and B. Falcidieno. The problem of form feature classification and recognition in CAD/CAM. *NATO ASI Series F*, 40:899–919, 1988.

[AFF85] S. Ansaldi, L. De Floriani, and B. Falcidieno. Geometric modelling of solid objects by using a face adjanceny graph representation. *Computer Graphics (Proceedings of Siggraph 1985)*, 19(3):131–139, 1985.

[AHU82] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1982.

[All97] J. A. Allderidge. Applying the definitive assembler maintainer (DAM) architecture to graphics. Third year undergraduate project report, Department of Computer Science, University of Warwick, May 1997.

[ANM96] A. L. Ames, D. R. Nadeau, and J. L. Moreland. *The VRML Sourcebook.* John Wiley and Sons Inc., 1996.

[APS96] V. D. Adzhiev, A. A. Pasko, and A. V. Sarkisov. "HyperJazz" project: development of geometric modelling systems with inherent symbolic interactivity. In *Proceedings of CSG 1996, Winchester*, pages 183–198. Information Geometers, April 1996.

[Aut92a] Autodesk Incorporated, Rue du Puits-Godet 6, 2000 Neuchatel, Switzerland. *AutoCAD Release 12 - Reference Manual*, 1992. Part number 600001-00104-09.

[Aut92b] Autodesk Incorporated, Rue du Puits-Godet 6, 2000 Neuchatel, Switzerland. *AutoLISP Reslease 12 - Programmer's Reference*, 1992. Part number 600002-00104-09.

[BACY94a] W. M. Beynon, V. D. Adzhiev, A. J. Cartwright, and Y. P. Yung. An agent-oriented framework for concurrent engineering. In *Proc. IEE Colloquium: Issues of co-operative working in concurrent engineering, Digest 1994/177*, pages 9/1–9/4, October 1994.

[BACY94b] W. M. Beynon, V. D. Adzhiev, A. J. Cartwright, and Y. P. Yung. A computational model for multiagent interaction in concurrent engineering. In *Proceedings of CEEDA 1994, Bournemouth University*, pages 227–232, 1994.

[Bar84] A. Barr. Global and local deformations of solid primitives. *ACM Computer Graphics*, 18(3):21–30, 1984. Proceedings of SIGGRAPH 1984.

[BB98] J. Berchtold and A. Bowyer. Bezier surfaces in set-theoretic geometric modelling. In *Proceedings of CSG 1998*. Information Geometers, 1998.

[BBCG+97] J. Bloomenthal, C. Bajaj, M. Cani-Gascuel, A. Rockwood, B. Wyvill, and G. Wyvill. *Introduction to Implicit Surfaces*. The Morgan Kaufmann Series in Computer Graphics and Geometric Modelling. Morgan Kaufmann, 1997.

[BBY92] W. M. Beynon, I. Bridge, and Y. P. Yung. Agent-oriented modelling for a vehicle cruise control system. In *Proceddings of ESDA 1992*, pages 159–165, 1992.

[BC88] W. M. Beynon and A. J. Cartwright. A defintive notation for geometric modelling. In *Proceedings of the 2nd Eurographics ICAD Workshop, CWI Amsterdam*, April 1988. Conceived as appendix to *A definitive approach to the implementation of CAD software.*

[BC89] W. M. Beynon and A. J. Cartwright. A definitive programming approach to the implementation of CAD software. In *Intelligent CAD Systems II: Implementation Issues*, pages 126–145. Springer—Verlag, 1989. Appendix published separately.

[BC92] W. M. Beynon and A. J. Cartwright. Enhancing interaction in computer aided design. In *Proc. Design and Automation Conference, HK*, pages 643–648, August 1992.

[BC95] W. M. Beynon and R. I. Cartwright. Empirical modelling principles for cognitive artefacts. In *Proceedings of IEE Colloquium: Design Systems* *with Users in Mind: The Role of Cognitive Artefacts*, pages 8/1–8/8. IEE Digest No. 95/231, December 1995.

[BC97] W. M. Beynon and R. I. Cartwright. Empirical modelling principles in application development for the disabled. In *Proceedings of IEE Colloquium Computers in the Service of Mankind: Helping the Disabled*, March 1997.

[BCCY97] W. M. Beynon, R. I. Cartwright, A. J. Cartwright, and Y. P. Yung. Abstract geometry for design in an empirical modelling context. Research Report 319, Department of Computer Science, University of Warwick, 1997.

[BCJ+95] A. Bowyer, S. Cameron, G. Jared, R. Martin, A. Middleditch, M. Sabin, and J. Woodwark. *Introducing Djinn. A geometric interface for solid modelling.* The Geometric Modelling Society and Information Geometers, 1995.

[BCJ+97] A. Bowyer, S. Cameron, G. Jared, R. Martin, A. Middleditch, M. Sabin, and J. Woodwark. Ten questions that arose in designing the Djinn API for solid modelling. In *Proceedings of International Conference on Shape Modelling and Applications, Aizu-Wakamatsu, Japan*, pages 71–76. IEEE Computer Society Press, March 1997.

[Bey83] W. M. Beynon. A definition of the ARCA notation. Research Report 54, Department of Computer Science, University of Warwick, 1983.

[Bey85] W. M. Beynon. Definitive notations for interaction. In Johnson and Cook, editors, *Proceedings of HCI 1985*, pages 23–34. Cambridge University Press, 1985.

[Bey86a] W. M. Beynon. ARCA - a notation for displaying and manipulating combinatorial diagrams. Research Report 78, Department of Computer Science, University of Warwick, 1986.

[Bey86b] W. M. Beynon. The LSD notation for communicating systems. Research Report 87, Department of Computer Science, University of Warwick, 1986. Presented at 3rd BCTCS, Leicester 1987.

[Bey89] W. M. Beynon. Evaluating definitive principles for interaction in graphics. In *New Advances in Computer Graphics, Proceedings of CG International*, pages 291–303. Springer-Verlag Tokyo, 1989. Also Reserch Report 133, Department of Computer Science, University of Warwick.

[Bey97] W. M. Beynon. Empirical modelling for educational technology. In *Proceedings of Cognitive Technology 1997*, pages 54–68. University of Aizu, Japan, IEEE, 1997.

[Bey98a] W. M. Beynon. Empirical modelling and the foundations of artificial intelligence. In *Proceedings International Workshop on Computation, Metaphor, Analogy and Agents*, 1998. To appear.

[Bey98b] W. M. Beynon. Modelling state in mind and machine. Research Report 337, Department of Computer Science, University of Warwick, February 1998.

[Bir91] S. Bird. An implementation of the ARCA translator. Third year undergraduate project report, Department of Computer Science, University of Warwick, 1991.

[BJ94] W. M. Beynon and M. Joy. Computer programming for noughts and crosses: New frontiers. In *Proceedings of PPIG 1994*, pages 27–37. Open University, January 1994.

[Bow93] D. S. Bowers. *From Data to Database.* Chapman and Hall, second edition, 1993.

[Bow94] A. Bowyer. *sVLIs Set—theoretic kernel modeller.* Information Geometers, 1994.

[Bra79] I. C. Braid. Notes on a geometric modeller. CAD Group Document 101, Computer Laboratory, University of Cambridge, 1979.

[Bro82] C. M. Brown. PADL-2: A technical summary. *IEEE Computer Graphics and Applications*, 2(2):69–84, March 1982. Includes *A tale of technical transfer* by Herb Voelcker.

[Brö95] P. Brödner. The two cultures in engineering. In *Skill, Technology and Enlightenment: On Practical Philosophy*, pages 249–260. Springer-Verlag, 1995.

[BSY95] W. M. Beynon, C.J. Sidebotham, and Y. P. Yung. Computer—assisted jigsaw construction: a case-study in empirical modelling. In *Proc. 5th Eurographics Workshop on Programming Paradigms for Graphics*, pages 37–50, September 1995.

[BY88a] W. M. Beynon and Y. W. Yung. Implementing a definitive notation for interactive graphics. In *New Trends in Computer Graphics*, pages 456–468. Springer—Verlag, 1988. Also Research Report 111, Department of Computer Science, University of Warwick.

[BY88b] W. M. Beynon and Y. W. Yung. Implementing a definitive notation for interactive graphics. In *New Trends in Computer Graphics*, pages 456–468. Springer-Verlag, 1988.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremacs. *Object-Oriented Development, The Fusion Method.* Object-oriented series. Prentice-Hall, 1994.

[Car94a] A. J. Cartwright. *Application of Definitive Scripts to Computer-Aided Conceptual Design.* PhD thesis, Department of Engineering, University of Warwick, July 1994.

[Car94b] R. I. Cartwright. A definitive approach to solid modelling computer aided design encapsulating R-functions. Third year undergraduate project report, Department of Computer Science, University of Warwick, 1994. Contains a description of the *CADNORT* notation.

[CB91] D. E. Carter and B. S. Baker. *Concurrent Engineering: The Product Development Environment for the 1990s.* Addison-Wesley, 1991.

[CH96] G. Cornell and C. S. Hortsmann. *Core Java.* SunSoft Press, 1996.

[Cor97] J. Corney. *3D Modelling with the ACIS Kernel and Toolkit.* Wiley, 1997.

[Cox65] H. S. M. Coxeter. *Generators and relations for discrete groups.* Ergebnisse der Mathematik und ihrer Grenzgebeite. Springer, second edition, 1965.

[CS98] H. Custer and D. Solomon. *Inside Windows NT.* Microsoft Press, second edition, 1998.

[CW89] M. Chmilar and B. Wyvill. A software architecture for integrated modelling and animations. In *New Advances in Computer Graphics*, pages 257–276. Springer-Verlag, 1989.

[Dep92] Department of Computer Science, University of Warwick. *Technical Document for the Scout System*, October 1992.

[Far90] G. Farin. *Curves and Surfaces in Computer-Aided Geometric Design: A Practical Guide.* Academic Press, second edition, 1990.

[FBMB90] B. N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, 1990.

[Fla96] D. Flanagan. *Java in a Nutshell.* O'Reilly and Associates, first edition, 1996.

[Fur96] S. Furber. *ARM System Architecture.* Addison Wesley Longman, 1996.

[FvF$^+$93] J. D. Foley, A. van DAM, S. K. Feiner, J. F. Hughes, and R. L. Phillips. *Introduction to Computer Graphics.* Addison-Wesley, 1993.

[GS93] A. Gibbons and P. Spirakis. *Lectures on Parallel Computation.* Cambridge International Series on Parallel Computation: 4. Cambridge University Press, 1993.

[GYC$^+$96] D.K. Gehring, S. Yung, R. I. Cartwright, W. M. Beynon, and A. J. Cartwright. Higher-order constructs for interactive graphics. In *Proceedings of the 14th Annual Conference of the Eurographics UK Chapter*, pages 179–192, 1996.

[Har87a] D. Harel. *Algorithmics.* Addison-Wesley, 1987.

[Har87b] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, July 1987.

[Har88] D. Harel. On visual formalisms. *CACM*, 31(5):514 – 530, May 1988.

[Har92] D. Harel. Biting the silver bullet: Towards a brighter future for software development. *IEEE Computer*, January 1992.

[Har97] E. R. Harold. *Java Network Programming.* The Java Series. O'Reilly and Associates Inc., first edition, February 1997.

[IEE85] Institute of Electrical and Electronics Engineers Inc., 345 East 47th Street, New York, NY 10017. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985 (IEE 754)*, 1985.

[Jac97] R. Jacobson. *Microsoft Excel 97 Step by Step.* Microsoft Press, August 1997.

[Joy94] M. S. Joy. *Beginning UNIX.* Tutorial Guides in Computing and Information Systems. Chapman and Hall, first edition, 1994.

[KCY93] A. Kaufman, D. Cohen, and R. Yagel. Volume graphics. *IEEE Computer*, 23(7):51–64, 1993.

[KDS96] C. Kinata, M. Dodge, and C. Stinson. *Running Microsoft Excel 97.* Microsoft Press, December 1996.

[KM96] S. Krishnan and D. Manocha. Efficient representations and techniques for computing B-reps of CSG models with NURBS primitives. In *Proceedings of CSG 1996*. Information Geometers, 1996.

[Knu68] D. E. Knuth. *The Art of Computer Programming*, volume 1, Fundamental Algorithms of *Addison-Wesley series in computer science and information processing.* Addison-Wesley, second edition, 1968.

[Kon92] K. Kondo. Algebraic method for manipulation of dimensional relationships in geometric models. *Computer-Aided Design*, 24(3):141–147, 1992.

[Kro94] E. Krol. *The Whole Internet User's Guide and Catalog.* O'Reilly and Associates, second edition, 1994.

[KSW86] F. Kimura, H. Suzuki, and L. Wingard. A uniform approach to dimensioning and tolerancing in product modelling. In *Computer Applications in Production and Engineering (CAPE 1986)*, pages 165–178. Elsevier Science, 1986.

[Lar94] M. H. E. Larcombe. Generalised distance functions and plastic modelling over and armature. In A. Bowyer, editor, *Computer-Aided Surface Geometry and Design: The Matematics of Surfaces IV*, pages 281–298. Cambridge University Press, 1994.

[LC87] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *ACM Computer Graphics*, 21(4):163–169, Jul 1987. Proceedings of SIGGRAPH 1987.

[LMB92] J. Levine, T. Mason, and D. Brown. *lex & yacc.* O'Reilly and Associates, second edition, 1992.

[Mac97] A. MacDonald. Physically based modelling. Third year undergraduate project report, Department of Computer Science, University of Warwick, 1997.

[MD97] J. Meyer and T. Downing. *Java Virtual Machine.* O'Reilly and Associates, 1997.

[MK97] C. Musciano and B. Kennedy. *HTML — The Definitive Guide*. O'Reilly and Associates, second edition, 1997.

[MPS96] K. T. Miura, A. A. Pasko, and V. V. Savchenko. Paramteric patches and volumes in function representation of geometric solids. In *Proceedings of CSG 1996, Winchester*, pages 217–231. Information Geometers, April 1996.

[Nar93] B. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, 1993.

[NBY94] P.E. Ness, W. M. Beynon, and Y. P. Yung. Applying agent-oriented design to a sail boat simulation. In *Proceedings of ESDA 1994*, volume 6, pages 1–8, 1994.

[Nes97] P. E. Ness. *Creative Software Development — An Empirical Modelling Framework*. PhD thesis, Department of Computer Science, University of Warwick, April 1997.

[ONK73] N. Okino, Y. Nakazu, and H. Kubo. TIPS-1: Technical information processing system for computer-aided design, drawing and manufacture. In J. Hatvany, editor, *Computer Languages for Numerical Control*, pages 141–150. North-Holland, 1973.

[Org98] Internaional Standards Organisation. Standard for exchange of product model data STEP. ISO 10303, 1998.

[Ous90] J. K. Ousterhout. Tcl: An embeddable command langauge. In *Proceedings of Winter USENIX*, pages 133–146, 1990.

[Ous91] J. K. Ousterhout. An X11 toolkit based on the Tcl language. In *Proceedings of Winter USENIX*, pages 105–116, 1991.

[PASS95] A. A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko. Function representation in geometric modelling: Concepts, implementation and application. *The Visual Computer*, 11(8):429–446, 1995.

[PFTV92] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, second edition, 1992.

[PPV95] A. Paoluzzi, V. Pascucci, and M. Vicentino. Geometric programming: A programming approach to geometric design. *ACM Transactions on Graphics*, 14(3):266–306, July 1995.

[PSA93] A. A. Pasko, V. Savchenko, and V. D. Adzhiev. Education project: Empty case technology of geometric modelling. In *Combined Proceedings of EDUGRAPHICS and COMPUGRAPHICS*, pages 6–11. ACM, 1993.

[Req80] A. A. G. Requicha. Representations for rigid solids: Theory, methods and systems. *ACM Computing Surveys*, 12(4):437–464, December 1980.

[Rum91] J. Rumbaugh. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Rus97] S. B. Russ. Empirical modelling: the computer as a modelling medium. *BCS Computer Buletin*, pages 296–301, April 1997.

[SC96] International Organization For Standardization and International Electrotechnical Commission. The virtual reality modelling language, version 2.0. Draft Specification ISO/IEC CD 147720, ISO/IEC, August 1996. Available at `http://vrml.sgi.com/moving-worlds/spec/`.

[Ser87] D. Serrano. *Constraint Management in Conceptual Design*. PhD thesis, Department of Engineering, MIT, 1987.

[She87] S. Sheridan. The scions of a solids modeler move into industry. *Mechanical Engineering*, pages 36–41, March 1987.

[Sla90] M. Slade. Definitive parallel programming. Master's thesis, Department of Computer Science, University of Warwick, April 1990.

[SM95] J. J. Shah and M. Mäntylä. *Parametric and Feature-Based CAD/CAM — Concepts, Techniques and Applications*. John Wiley and Sons, 1995.

[SP94] V. M. Savchenko and A. A. Pasko. Shape transformations of 3D geometric solids. In *Proceedings of International Workshop on Shape Modelling: Parallelism, Interactivity and Applications*, pages 47–53. University of Aizu, Japan, September 1994.

[Spi92] J. M. Spivey. *The Z Notation*. International series in Computer Science. Prentice Hall, 1992.

[SPS96] A. Sourin, A. Pasko, and V. Savchenko. Using real functions with application to hair modelling. *Computers and Graphics*, 20(1):11–19, 1996.

[Sti89] J. M. Stidwill. The CADNO programming language. Third year undergraduate project report, Department of Computer Science, University of Warwick, May 1989.

[Suc87] L. A. Suchman. *Plans and Situated Actions: The Problem of Human-machine Communication*. Learning in doing: Social, cognitive, and computation perspectives. Cambridge University Press, 1987.

[Sut63] I. E. Sutherland. A man-machine graphical communication system. In *Spring Joint Computer Conference*, volume 23, 1963.

[Tea] The Empirical Modelling Project Team. The empirical modelling world-wide-web site. `http://www.dcs.warwick.ac.uk/modelling/`.

[Tom89] T. Tomiyama. Meta-model: a key to intelligent CAD systems. *Research in Engineering and Design*, 1(1):19–34, 1989.

[Vee92] P. Veerkemp. *On the Development of an Artefact and Design Description Language*. PhD thesis, CWI Amsterdam, 1992.

[vERR93] M. van Emmerick, A. Rappoport, and J. Rossignac. Simplifying interactive design of solid models: a hypertext approach. *The Visual Computer*, 9:239–254, 1993.

[VRH+78] H. B. Voelcker, A. A. G. Requicha, E. E. Hartquist, W. B. Fisher, J. Metzger, R. B. Tilove, N. K. Birell, W. A. Hunt, G. T. Armstrong, T. F. Check, R. Moote, and J. McSweeney. The PADL-1.0/2 system for defining and displaying solid objects. *Computer Graphics (Proceedings of Siggraph 1978)*, 12(3):257–263, August 1978.

[Weg97] P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, May 1997.

[WG97] B. Wyvill and A. Guy. The blob tree implicit modelling and VRML. In *From the Desktop to the Webtop: Virtual Environments on the Internet, WWW and Networks, NMPFT, Bradford*, 1997.

[WND97] M. Woo, J. Neider, and T. Davis. *The OpenGL Programming Guide*. Addison-Wesley, second edition, 1997. In two volumes: *The official guide to learning OpenGL 1.1* and *OpenGL Reference Manual*.

[WvO95] B. Wyvill and K. van Overveld. Constructive soft geometry: The unifications of CSG and implicit surfaces. Technical report, Department of Computer Science, University of Calgary, April 1995.

[WvO97] B. Wyvill and K. van Overveld. Warping as a modelling tool for CSG / implicit models. In *Proceedings of International Conference on Shape Modelling and Applications, Aizu-Wakamatsu, Japan*, pages 205–213. IEEE Computer Society Press, March 1997.

[Wyv75] B. Wyvill. *An Interactive Graphics Language*. PhD thesis, Bradford University, 1975.

[Yun90] Y. W. Yung. EDEN: An engine for definitive notations. Master's thesis, Department of Computer Science, University of Warwick, September 1990.

[Yun93] Y. P. Yung. *Definitive Programming: A Paradigm for Exploratory Programming*. PhD thesis, Department of Computer Science, University of Warwick, January 1993.

[Yun96] Y. P. Yung. Agent-oriented modelling for interactive systems. Report at the end of EPSRC grant GRJ13458., Department of Computer Science, University of Warwick, July 1996.

[YY88] Y. P. Yung and Y. W. Yung. *The EDEN Handbook*. Department of Computer Science, University of Warwick, 1988. Last updated in 1996.