

AUTHOR: Ashley Thomas Ward DEGREE: Ph.D.

**TITLE: Interaction with Meaningful State:
Implementing Dependency on Digital Computers**

DATE OF DEPOSIT:

I agree that this thesis shall be available in accordance with the regulations governing the University of Warwick theses.

I agree that the summary of this thesis may be submitted for publication.

I **agree** that the thesis may be photocopied (single copies for study purposes only).

Theses with no restriction on photocopying will also be made available to the British Library for microfilming. The British Library may supply copies to individuals or libraries, subject to a statement from them that the copy is supplied for non-publishing purposes. All copies supplied by the British Library will carry the following statement:

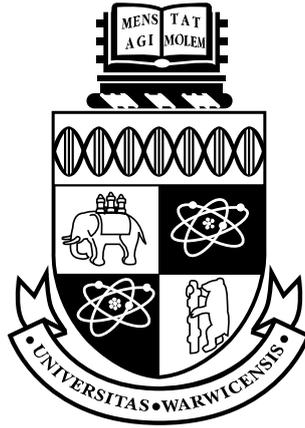
“Attention is drawn to the fact that the copyright of this thesis rests with its author. This copy of the thesis has been supplied on the condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the author’s written consent.”

AUTHOR’S SIGNATURE:

USER’S DECLARATION

1. I undertake not to quote or make use of any information from this thesis without making acknowledgement to the author.
2. I further undertake to allow no-one else to use this thesis while it is in my care.

DATE	SIGNATURE	ADDRESS
.....
.....
.....
.....
.....



**Interaction with Meaningful State:
Implementing Dependency on Digital Computers**

by

Ashley Thomas Ward

Thesis

Submitted to the University of Warwick

for the degree of

Doctor of Philosophy

Department of Computer Science

May 2004

THE UNIVERSITY OF
WARWICK

*To Uncle Adam, Uncle Colin,
Mr Cheal, Mikki Larcombe
and Meurig with thanks*

Contents



List of Figures	iv
List of Tables	viii
List of Listings	ix
Acknowledgements	xi
Declaration	xii
Abstract	xiii
Abbreviations	xiv
Chapter 1 Introduction	1
1.1 Dependency in concept: EM, Radical Empiricism and the making of meaning	3
1.2 Dependency in application: Interaction with Meaningful State	10
1.3 Dependency in development: a novel abstraction	19
1.3.1 Modelling requirements	21
1.3.2 Program comprehension and validation through abstraction	22
1.4 Dependency in engineering: simple and consistent relationships	26
1.5 Related work	32
1.6 Thesis aims	41
1.7 Thesis outline	43
Chapter 2 The Abstract Definitive Machine, ADM	46
2.1 Concepts of the LSD notation and the ADM	47
2.1.1 LSD accounts	47
2.1.2 1-agent modelling with the ADM	52
2.1.3 ‘Programming’ the ADM	54
2.1.4 From LSD accounts to ADM scripts	58
2.2 Implementations of the ADM	60
2.2.1 Evaluation/storage implementation strategies for definitive systems	60
2.2.2 Existing implementations of the ADM	61
2.2.3 A hardware implementation?	63
2.3 Operational semantics	64
2.3.1 Operational semantics of LSD	64

2.3.2	Invalid transitions in the ADM	67
2.3.3	What's in a transition?	69
2.3.4	Divisible command lists and the 'Authentic' ADM (AADM)	74
2.4	The ADM and UNITY	81
2.5	Background/Sources	88
2.A	Using the <code>am</code> implementation	91
Chapter 3 The Definitive Assembly Maintainer (DAM) machine		97
3.1	The DMM and the BRA	99
3.1.1	State and the DMM	99
3.1.2	Transitions and the block redefinition algorithm (BRA)	103
3.1.3	Efficiency and generality of the DMM	107
3.2	The DAM machine, from the bottom up	109
3.2.1	The platform basis for the DAM machine	110
3.2.2	Values	112
3.2.3	Operators	112
3.2.4	Data structure	113
3.2.5	DAM execution	117
3.2.6	External agency	120
3.3	!Donald	123
3.3.1	!Donald overview	124
3.3.2	DoNaLD to DAMscript	127
3.3.3	DAMscript to DAM data structure	130
3.3.4	Graphical action execution in !Donald	131
3.4	Definitive programming in !Donald2	140
3.4.1	Extending !Donald	140
3.4.2	Using raw DAMscript: the parabola example	141
3.4.3	DAM machine operators in BASIC and the dereference problem	149
3.4.4	!Donald performance	155
3.5	Geometry of definitions in the DAM machine store	162
3.5.1	Dependency between multi-word data types	162
3.5.2	Dependency on variable length data	163
3.5.3	Visualising the store	166
3.5.4	Exploiting the visualisation	170
3.A	The Script Digraph	178
Chapter 4 The Engine for Definitive Notations, EDEN		194
4.1	EDEN, chronologically to 1999	196
4.1.1	The early history	196
4.1.2	Formula variables and actions	199
4.1.3	<i>texteditorYung1987</i>	201
4.1.4	DoNaLD, SCOUT and ARCA pipeline translators	209
4.1.5	The early <code>tkeden</code>	214
4.1.6	Distributed <code>tkeden</code> : <code>dtkeden</code>	216

4.1.7	Fifty versions: an overview	218
4.2	Developments to EDEN since 1999	223
4.2.1	The beginnings: Sasami, motivating “in itself”	223
4.2.2	A database in EDEN: EDDI	229
4.2.3	A front-end parser in EDEN: the Agent Oriented Parser	231
4.2.4	Generalising the parser solution: the notations framework	239
4.2.5	Making the parser dependency driven?	240
4.2.6	Demonstrating EDEN with presentations	242
4.2.7	Interface improvements	245
4.2.8	Novel interfacing	250
4.3	EDEN execution and scheduling	259
4.3.1	Terminology mostly explicitly dismissed	260
4.3.2	Requirements of an EDEN-like definition maintainer	262
4.3.3	Black-box analysis of EDEN	264
4.3.4	Differences between definitions and actions in Eden	269
4.3.5	Inside the machine: definitions and actions	269
4.3.6	The EDEN scheduler algorithm	270
4.3.7	Higher Order Definitions (HODs) in EDEN	275
4.A	Pseudo-code of the EDEN scheduler algorithm	277
Chapter 5 Problematic issues in dependency maintenance		280
5.1	Concurrent definition maintenance	280
5.1.1	Mapping evaluation agency to definition-agents	281
5.1.2	Synchronisation of concurrent definition-agents	287
5.2	Moding	294
5.2.1	Problems with definitive lists	294
5.2.2	Meziani’s DENOTA and the “mode of definition” of a variable	301
5.3	The tri-box framework for higher-order definitive state	303
5.3.1	Motivation	304
5.3.2	Concept and examples	309
5.3.3	Implementation issues	321
5.A	Concurrent definition maintenance in SR	328
Chapter 6 Conclusions, contributions and future work		343
6.1	Conceptual machine models for EM	344
6.2	Empirical study of the EM tools and background scholarship	345
6.3	Identification of subtleties associated with dependency	347
6.4	Strengthening connections with other work	348
6.5	Future work: dependable computing, psychology and foundations	349
Bibliography		353
Index		372
Colophon		386

List of Figures

1.1	An Ishikawa fishbone diagram	5
1.2	The Empiricist Framework for Learning	7
1.3	The Phillips machine in the London Science Museum	9
1.4	A physical planimeter shown in the London Science Museum	13
1.5	Charles Care's Planimeter ISM	13
1.6	The Temposcope ISM	15
1.7	The digital watch ISM	15
1.8	A spreadsheet program and EDEN, showing observables, dependency and agency	17
1.9	Synchronisation techniques and language classes	24
1.10	A digital combinatorial circuit, definitive script and 'i/o equivalent' computer program	27
1.11	A single gate, compared with 'i/o equivalent' computer program	29
1.12	A 'pulse generator' combinatorial circuit where propagation time is significant	31
1.13	Count of articles including the word 'spreadsheet' in the title	35
2.1	BNF describing syntax for an LSD account	49
2.2	LSD account of a train carriage door and passenger	51
2.3	BNF describing syntax for the ADM	56
2.4	D , A and P in the ADM	57
2.5	Sequential <code>am</code> implementation and parallel conception of the ADM	74
2.6	Excerpts from Y.P. Yung's railway passenger LSD account and ADM script	75
2.7	Syntax for redefinition and evaluation from three implementations of the ADM	76
2.8	Partial BNF for UNITY	82
3.1	Architecture of an Acorn Archimedes 400 series computer, predecessor of the Risc PC	111
3.2	DAM machine data representation and interface routines	115
3.3	State transitions in the DAM machine	118
3.4	A DAM machine operator cleaning partial state	120
3.5	Overview of !Donald application components	125
3.6	The !Donald application	126

3.7	Compilation, Loading and “Execution” in !Donald	128
3.8	Selecting a !Donald viewport	134
3.9	A screen shot of !Donald with Listing 3.3 loaded, showing the DAM store	134
3.10	DAM machine data structure shown in Figure 3.9	136
3.11	The !Donald2 user interface (compare Figure 3.6, p.126)	141
3.12	A screenshot of the parabola	145
3.13	The “off screen” label in the parabola DAMscript	147
3.14	Graphical display from “Dereference” DAMscript, before and after a change to <code>yspace</code>	152
3.15	“Dereference” after the change to <code>name1</code>	153
3.16	“Dereference” after the change to <code>a</code>	154
3.17	Structure of the “numeric” script	160
3.18	DAM machine operator configuration for summation with double length data types	163
3.19	Visualisation of the “parabola” DAMscript DAM store	167
3.20	Visualisation of the “engine” DoNaLD script DAM store	168
3.21	DAM store mapping to pixels	169
3.22	Lookup of an ASCII character code glyph	173
3.23	Spreadsheet form of “text” DAMscript	175
3.24	Visualisation of the “text” DoNaLD script DAM store	176
3.25	Simple graph cycles	179
3.26	<code>xvcg</code> visualisation of <i>roomviewerYung1991</i>	181
3.27	Wong’s Dependency Modelling Tool, showing the ATM script	182
3.28	Acyclic digraphs graphically enumerated, $3 \leq N \leq 5$	187
4.1	<i>texteditorYung1987</i> running in <code>ttyeden-1.52</code>	201
4.2	<i>texteditorYung1987</i> with some example redefinitions	209
4.3	The original DoNaLD system	211
4.4	ARCA, SCOUT and DoNaLD combined on a pipeline	212
4.5	<code>tkyeden-dec151997</code> running <i>cruisecontrolBridge1991</i>	215
4.6	<code>tkyeden</code> architecture	215
4.7	“God’s eye view” in <i>claytontunnelSun1999</i> on the <code>dtyeden</code> server	217
4.8	<code>dtyeden</code> and <i>claytontunnelSun1999</i> in use with school children during the ACE week in January 1999	218
4.9	Code size of EDEN from 1988 to 2004 (1984 <code>hoc</code> shown for comparison)	219
4.10	<code>tkyeden</code> running on Solaris, Linux, Windows and Mac OS X	221
4.11	Data from logging of local <code>tkyeden</code> usage and downloads	222
4.12	The Sasami Rubik’s cube, <i>rubiksCarter1999</i>	224
4.13	Data flow in Sasami	226
4.14	The effect of changing the <code>small_cube_size</code> variable in <i>rubiksCarter1999</i>	226
4.15	EDDI syntax	230
4.16	AOP EDDI to EDEN translation	234

4.17	Observational structure of the AOP	236
4.18	Harfield’s parser builder in <i>agentparserHarfield2003</i>	237
4.19	Harfield’s parser visualisation in <i>agentparserHarfield2003</i>	238
4.20	<code>tkeden</code> input window with a selection of the multiple notations now available installed	239
4.21	Notation input data flow in EDEN	241
4.22	The first presentation environment slide	242
4.23	The ‘jugs’ model in a presentation slide	244
4.24	<code>tkeden</code> interface changes: <code>tkeden-dec151997</code> contrasted with <code>tkeden-1.46</code>	246
4.25	<code>tkeden-1.46</code> menus	246
4.26	Part of a simplified <code>tkeden</code> interface constructed in SCOUT	248
4.27	Sasami window linked to a USB steering wheel using a definition	252
4.28	Jon McHale and <i>carparkingsimMcHale2003</i>	252
4.29	Chris Rose and the model railway hardware	254
4.30	A partial screenshot of <i>modelrailwayRose2004</i>	255
4.31	Communication between EDEN and an Interactive Process notation translator	258
4.32	Requirements of an EDEN-like definition maintainer	263
4.33	Depth- and breadth- first topological sorts	264
4.34	Definitive script graphs containing only definitions systematically con- structed to examine DM evaluation strategies	266
4.35	An equivalent definition and action	267
4.36	Definitive script graphs containing only actions systematically trans- formed from Figure 4.34, which contains only definitions	268
4.37	EDEN machine data, operations, data flow and control flow	272
4.38	The EDEN machine states considered more statically	273
5.1	The script graph for <code>a is b+c</code> , together with an associated extended script graph devised by adding a definition-agent node	282
5.2	Two possible definition-agent arrangements for a simple two-line defin- itive script	283
5.3	A script graph, the corresponding “monolithic” and one-to-one con- figurations of definition-agent nodes	284
5.4	Decomposition of the polynomial definition $f = ax^3 + bx^2 + cx + d$ into two possible definitive scripts	286
5.5	Synchronisation for indivisible observation of state in the single defin- ition case	290
5.6	EDEN implements lists “horizontally” in the symbol table	298
5.7	Transformations implemented by the <code>%edens1</code> translator	299
5.8	The <code>%edens1</code> translator transforms references to lists, causing EDEN to locate them “vertically”	300
5.9	O-, C- and U-agents interacting with the state S	304
5.10	O-, C- and U-agents observe & act on subsets of S and are synchronised	308
5.11	Script graph and tri-box diagram for <code>a is b+c</code>	309

5.12	Tri-box diagrams of the “power” script	312
5.13	Tri-box diagram showing an on-screen character glyph representation linked by dependency to a box containing a character code	314
5.14	Two U-agents observing sub-sets of a list	316
5.15	An <code>if</code> conditional HOD in the tri-box framework	318
5.16	Two O-agents perceiving different dependencies	320
6.1	Dependency and agency in the current EM tool architecture	350
6.2	Spectrum of empirical work	351

List of Tables

2.1	LSD and ADM terminology compared	59
3.1	An example script in LLDN	100
3.2	Numerical !Donald2 operators	142
3.3	Geometrical !Donald2 operators	142
3.4	Miscellaneous !Donald2 operators	142
3.5	<code>tkeden</code> and !Donald performance compared when running on different machines	157
3.6	<code>tkeden</code> and !Donald performance compared when running on the <i>same</i> machine	157
3.7	<code>tkeden</code> and !Donald2 performance when running the “numeric” non-graphical script (Listing 3.9)	160
3.8	Number of acyclic digraphs with N unlabelled nodes	185

List of Listings

1.1	A description file for the <code>make</code> tool	38
2.1	Two ADM entities that can compute GCD	55
2.2	Slade’s ADM algorithm	69
2.3	An interaction with <code>am</code> demonstrating evaluation in <i>S</i> state	73
2.4	A proposed algorithm for ‘Authentic’ ADM (AADM) execution (Ward, after Beynon and Slade)	79
2.5	Y.P. Yung’s <code>am</code> cat flap script	92
2.6	An interaction with <code>am</code> and Y.P. Yung’s cat flap	96
3.1	ARM code for a DAM machine <code>add</code> operator	113
3.2	A simple DoNaLD script with multiple viewports	133
3.3	DAMscript translation of the DoNaLD script shown in Listing 3.2	133
3.4	ARM code for the <code>!Donald line</code> graphical action operator	138
3.5	The “parabola” DAMscript	144
3.6	Definitions relating to the “off screen” label, written in a fictional language	148
3.8	The BASIC functions used by the Listing 3.7 DAMscript	151
3.7	The “dereference” DAMscript	151
3.9	“Numeric” Eden script and DAMscript	159
3.10	A DAM machine ‘lookup’ operator	164
3.12	<code>chargraphic</code> BASIC function used by “text” DAMscript (Listing 3.11)	173
3.11	“text” DAMscript	173
4.1	An interaction with <code>hoc</code>	198
4.2	A selection of Read-Write Variables from <i>texteditorYung1987</i>	202
4.3	Some Formula Variables in <i>texteditorYung1987</i>	203
4.4	Some actions in <i>texteditorYung1987</i>	204
4.5	Some procedures in <i>texteditorYung1987</i>	205
4.6	Highlights from the <i>texteditorYung1987</i> <code>edit</code> procedure	206
4.7	Some possible redefinitions to make to <i>texteditorYung1987</i>	207
4.8	Eden output corresponding to the Sasami definition <code>vertex v (a+10)/20 -c 1.0</code>	225
4.9	Interaction with the <code>eddip</code> translator	232

4.10	EDEN script corresponding to part of Figure 4.22	243
4.11	EDEN script that instantiates the slide shown in Figure 4.22	244
4.12	Using the USB HID facilities in <code>tkeden</code>	251
4.13	Eden code to communicate with the train PIC	254
5.1	SR code implementing the synchronisation shown in Figure 5.5	291

Acknowledgements

Most of all, I would like to thank my supervisor Meurig Beynon for his trust, patience and inspiration. Mine has been a wonderful apprenticeship. Steve Russ has also played a major part in my supervision and I am deeply grateful for his thoughtful advice.

Many other people have also assisted with this thesis, directly and indirectly.

I would like to thank the selection of previous EM researchers whose work I have had the pleasure of studying closely for this thesis. If I criticise their work, I do so only in an attempt to stand on their shoulders. In chronological order of their contribution: Y.W. ‘Edward’ Yung, Samia Meziani, Mike Slade, Y.P. ‘Simon’ Yung, Dominic Gehring, James Allderidge, Pi-Hwa ‘Patrick’ Sun and Richard Cartwright. I am also very grateful to Richard for introducing me to this area.

I am pleased to acknowledge Russell Boyatt, Chris Brown, Ben Carter, “the other” Ashley Chaloner, Josie Clement, Richard Cunningham, Carlos Fischer, Nathan Griffiths, Antony Harfield, Tim Heron, Mark Lloyd, Ria Lloyd, Izellah Macintosh, Jon McHale, Rod Moore, Roger Packwood, Chris Roe, Chris Rose, Dorian Rutter, Stuart Valentine and Paul Williamson for technical assistance related to this thesis. I also thank my other fellow EM-ers for many thought-provoking seminars and models, as well as Abhir Bhalerao, Peter Forbrig, Katie Lucas, Steve Matthews, Simon Rawles, Hans Roeck, Matt Street and Rob West for useful input.

I would especially like to thank Chris Roe for his companionship as we explored this landscape together. Many thanks also to my trusted friend Benjohn Barnes for many wide-ranging discussions over the course of this work.

Since May 2002 I have been working part-time on this thesis — my Teaching Fellow ‘day job’ has involved the teaching of Computer Organisation and Architecture (COA). I am indebted to my COA colleagues Chang-Tsun Li, Franc Buxton, Gerry Biggs and TRM, RAP, SGV, RC, PW already mentioned above. I also thank my other colleagues and friends in the department. I particularly appreciate the generosity and trust of Graham Nudd.

I am extremely grateful to my family and friends for moral support. Thanks to Mum and Dad for (amongst many other things), instilling organisational skills and encouraging me to ask ‘why’.

Finally, thank you, Linda, for the unconditional love and support, providing the stability without which this thesis would never have been completed.

Thanks to my external examiners Mike Holcombe and Jean Bacon for the timely viva and helpful comments.

Declaration

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. All quoted text remains the copyright of the original attributed author. I have taken the liberty of correcting typographical errors, spelling and grammar where necessary. All trademarks are acknowledged.

None of the material in this thesis has been previously published, but some of the motivating ideas have appeared in joint work published in [BCSW99, BWM⁺00, BCH⁺01, BRWW01, BBRW03].

Interaction with Meaningful State:
Implementing Dependency on Digital Computers

Abstract

A perception of an agent as follows — that the act of changing the value of one observable indivisibly entails a predictable accompanying change in that of another — can be termed a *dependency*. Indivisibility in change allows us, as agents, to make our experience of interaction with the ‘world’ meaningful.

Empirical Modelling (EM) is the name we have given at Warwick to the activity of building artefacts to embody patterns of observables, dependencies and agent actions that are encountered in experience. EM involves the progressive development of understanding through interaction, whereby meaning is continually refined in the light of additional experience. An EM artefact can be physical (*cf.* the Phillips machine [Phi00]) or computer-based (*cf.* spreadsheets).

This thesis investigates how dependency can be effectively implemented on a digital computer through the critical evaluation of three contrasting tools to support EM developed at Warwick over the last 16 years. The thesis contribution has three aspects: conceptual insight; critical, historical and empirical review; and technical and practical development.

Slade’s original Abstract Definitive Machine (ADM) concept [Sla90] and its implementations are reviewed and linked to the original motivation for its development — animating LSD accounts. The main emphasis of the ADM is on agency. Subtleties and inconsistencies in the way in which the ADM concept has developed over time are exposed and an algorithm for an ‘Authentic’ ADM (AADM) is proposed.

Cartwright’s design and implementation of the Definitive Assembly Maintainer (DAM) machine [Car99] is analysed and critiqued. The DAM machine emphasises only dependency. The DAM machine is extended to allow lower-level interaction using a special-purpose code and to exploit the video hardware to create dependency at a very low level.

The primary tool of the EM research group, EDEN [Yun93, Yun90], successfully combines both dependency and agency. Its development is reviewed in its historical context and various extensions, including novel interfacing, are described. The first exposition of EDEN’s internal operation is given.

The analysis of existing EM tools motivates the examination of some problematic issues regarding concurrency, moding and higher-order dependency. Some proposals for design and implementation to address these issues are described.

Throughout the thesis, consideration of technical issues is motivated by the possibility that human engagement with computers can have qualities similar to engagement with ‘real-world’ artefacts, manifest in interaction with meaningful state that is at all times intelligible to the human interpreter, unlike the typically meaningless intermediate states that are generated in the execution of a conventional program. The insights that exploring this possibility brings may be significant in producing programs that are more robust under change.

Abbreviations

AADM	Authentic ADM
ADM	Abstract Definitive Machine [Sla90]
AOP	Agent-Oriented Parser [Har02, Bro01]
ARCA	A definitive notation named after Arthur Cayley (1821–1895), which “might be charitably regarded as ‘An Aid for the Realisation of Combinatorial Artefacts’” [Bey83]
BRA	Block Redefinition Algorithm [Car99] (my abbreviation)
CADNO ¹	Computer-Aided Design Notation ([Bey89b], first named in [Bey89a])
DAM machine	Definitive Assembly Maintainer machine [Car99]
DAMscript	A type of Low Level Definitive Notation used to program the DAM machine
DM	Dependency Maintainer
DMM	Dependency Maintainer Model [Car99]
!Donald	An implementation of DoNaLD running on the DAM machine
DoNaLD	Definitive Notation for Line Drawing [BABH86]
dtkeden	An implementation of EDEN that allows distributed communication [Sun99]
EDDI	Eden Definition Database Interpreter [Tru96]
EDEN	Engine for DEfinitive Notations [Yun90] — in this thesis, upper case ‘EDEN’ denotes an implementation
Eden	The language implemented by EDEN [Yun89]
EM	Empirical Modelling

¹Also Welsh for “fox”.

empublic	A local directory containing our tools installation and collected archive of models, partially accessible via the web [WRB]
FV	Formula Variable [Yun90] (in Eden)
HOD	Higher Order Dependency/Definition
ISM	Interactive Situation Model [Sun99]
LLDN	Low Level Definitive Notation [Car99] (my abbreviation)
LSD	Language for Specification and Description [Bey87a]
RWV	Read-Write Variable [Yun90] (in Eden)
S, S^*, S' states	States before, during and after a transition
Sasami	A definitive notation for EDEN allowing use of 3D graphics, implemented using OpenGL [Car00]
SCOUT	SCreen layOUT — a definitive notation [Yun93]
tkeden	An implementation [Yun93] of EDEN with a graphical user interface implemented using Tcl/Tk [Ous94]
ttyeden	An implementation of EDEN with a terminal-based ‘curses’ interface
UNITY	Unbounded Nondeterministic Iterative Transformations — a computational model and proof system [CM88]
VM	Virtual Machine

Chapter 1

Introduction



This thesis is motivated by my work for the Empirical Modelling (EM) research project, with which I have been closely associated for six and a half years. Over this period, I have witnessed a significant development in the scope of application and degree of student involvement in the project. This has been represented in the completion of eight PhD theses on a wide variety of themes, the emergence of a new fourth year undergraduate “Introduction to EM” module and some 40 third year undergraduate projects. I have had an explicit role through the co-authorship of several papers relating to applications [BCSW99, BWM⁺00, BCH⁺01, BRWW01, BBRW03] and the development of the models *catflapWard1997*¹, *emhttpdWard1999*, *musiccorexptWard1999*, *vcgWard1999*, *oasysprivilegesWard2000*, *definitivedmWard2001*, *lsmpresentationWard2001*, *backroomWard2002*, *blankpresentationWard2002*, *introtoempresentWard2002*, *sqleddiWard2003*, but yet more significant has been my role as the principal developer of EDEN², the primary tool of the EM research group, and as a major contributor to the management and dissemination of documentation and resources [WRB, Warb, Warc]. In particular, my extension and debugging of EDEN has contributed significantly to what is possible with the tools, and many of the more recent models could not have been constructed without it.

¹Throughout this thesis, text of the form *projectAuthorYear* refers to the unique key name of a project which can be found in our ‘empublic’ archive [WRB]. Further information about a project can be found at <http://empublic.dcs.warwick.ac.uk/projects/keyname>.

²From version 1.0 for Solaris in October 1999 to version 1.50 for Solaris, Linux, Windows and Mac OS X in March 2003 — see §4.2.

My experience of EM technical support and consultancy has led me to identify the *study of dependency* as one of the central issues in enabling EM. It has also informed a perspective on the current status of EM principles and tools that has both favourable and critical aspects:

Favourable — EM leads to a different quality of human-computer interaction, characteristic of “thinking with computers” [BR]. It has also become clear to me that EM also potentially provides a radically new approach to software development. Its distinctive nature gives support for change, comprehension and reuse. The use of EM principles may offer a promising route to software that is more dependable and intelligible. These qualities have both motivated and been informed by my practical work on the EM project, which has involved the development of parts of EDEN “in itself”, case studies in the use of dependency, and the development of new techniques for the maintenance and management of models.

Critical — My engagement with EM tools and models has also given me a complementary awareness of problematic issues that is deeper than that of a casual EM model builder. Designing and implementing EM tools exposes the tensions between human interpretation and machine implementation, challenges in documentation and version control and the difficulty of representing certain types of dependency.

My perspective on EM motivates the two main questions addressed in this thesis: How can we best exploit and develop our existing tools for implementing EM activity? What prospects are there for better tools in the future? The key issue in addressing these questions is the implementation of dependency on digital computers.

As a “second generation” thesis³ on Empirical Modelling, this thesis assumes a considerable body of underpinning knowledge about EM that cannot be made explicit within its scope. Because the bulk of the thesis is concerned with implementation issues, its focus is necessarily somewhat internal to the EM project. This accounts for the extended introduction that follows, which not only serves to introduce the

³And possibly the first such thesis.

technical contribution of the thesis but to provide essential EM background and to situate my research in relation to external literature.

The next four main subsections consider EM in relation to philosophy, applications, development and implementation. With specific reference to *dependency*, the key concept in this thesis, these subsections are respectively concerned with “dependency in concept”, “dependency in application”, “dependency in development” and “dependency in engineering”. Subsections describing related work, the thesis aims and outline then follow.

1.1 Dependency in concept: EM, Radical Empiricism and the making of meaning

Dependency is one of three primary concepts used in Empirical Modelling. This section explains these concepts, firstly without discussing their relationship to digital computers. The explanations inform the notion of *meaning* that is applied in this thesis. The concept of dependency plays a major part in the making of meaning.

The concepts of Empirical Modelling are based on a world-view similar to the Radical Empiricism of William James [Jam12]. Traditional empiricism is “the view that experience, especially of the senses, is the only source of knowledge” [her00]. Radical Empiricism goes further than traditional empiricism in its recognition that: “the relations that connect experiences must themselves be experienced relations, and any kind of relation experienced must be accounted as ‘real’ as anything else in the system” [Jam12, p.42]. To paraphrase Wild’s discussion of Radical Empiricism [Wil69], we must dwell on direct experience, vague and subjective though it is, and attempt to use concepts to clarify and express the implicit meanings present within it. When we approach a new problem, we must begin all over again, letting the direct experience speak, without forcing it into our prior established categories. We may have our own systems and conclusions, but must be ready to examine any new fact at any time and make the necessary revisions and corrections. There is always potentially more to learn, so facts have an element of ‘mystery’ and all conclusions are tentative [Wil69, pp.413, 390, 394–5].

Most scientific disciplines “begin with what is known by direct acquaintance”,

1.1. Dependency in concept: EM, Radical Empiricism and the making of meaning

but then “each of them leaves this behind, in order to turn to the special objects of its field, and to deal with them in as objective a way as is possible.” [Wil69, p.413]. For example, the discipline of Computer Science is largely concerned with the study of the Turing Machine, an abstract fictional object.

The Radical Empiricist world-view is appropriate in many fields concerned with analysis of phenomena. Accident investigations are one example. Section §4.1.6 illustrates how Empirical Modelling can be applied to the scenario of the Clayton Tunnel railway accident of 1861, as it is described in [Rol82]. The descriptions of the accident given by each participant are based on their situated, personal experience. Put together, the descriptions contain many gaps and logical inconsistencies. However it is still possible (in this case) to come to tentative conclusions about the probable cause of the accident. Further details are given in [Sun99, Bey99, BS99].

A simple type of analysis that can be performed in this kind of domain is illustrated by the ‘Cause and Effect’ diagram invented by Kaoru Ishikawa [IL85, p.63], nicknamed the ‘fishbone’ diagram due to its shape. Such a diagram is used to assist the identification of causal factors that contribute towards an undesirable quality characteristic (effect) observed within experience. The effect that is the subject of analysis is first recorded at one side of the diagram, then the ‘backbone’ of the fish is drawn and a systematic attempt is made to conceive potential causes. (When used in a manufacturing setting, categories of Man, Method, Material, Machine and Measurement are often used to assist in completeness.) Figure 1.1 contains an example (reproduced with permission from [Mor]) which shows possible causes of “missed free-throws” in a basketball tournament.

Analytical reduction, where problems are divided into distinct parts and then examined separately, is a common Computer Science technique for dealing with complexity. Checkland [Che99, p.59] states the limits on the appropriateness of reductionism.

Descartes’s second rule for ‘properly conducting one’s reason’, i.e. divide up the problems being examined into separate parts — the principle most central to scientific practice — assumes that this division will not distort the phenomenon being studied. It assumes that the components of the whole are the same when examined singly as when they are playing their part in the whole, or that the principles governing the assembling of the components into the whole are themselves straightforward.

1.1. Dependency in concept: EM, Radical Empiricism and the making of meaning

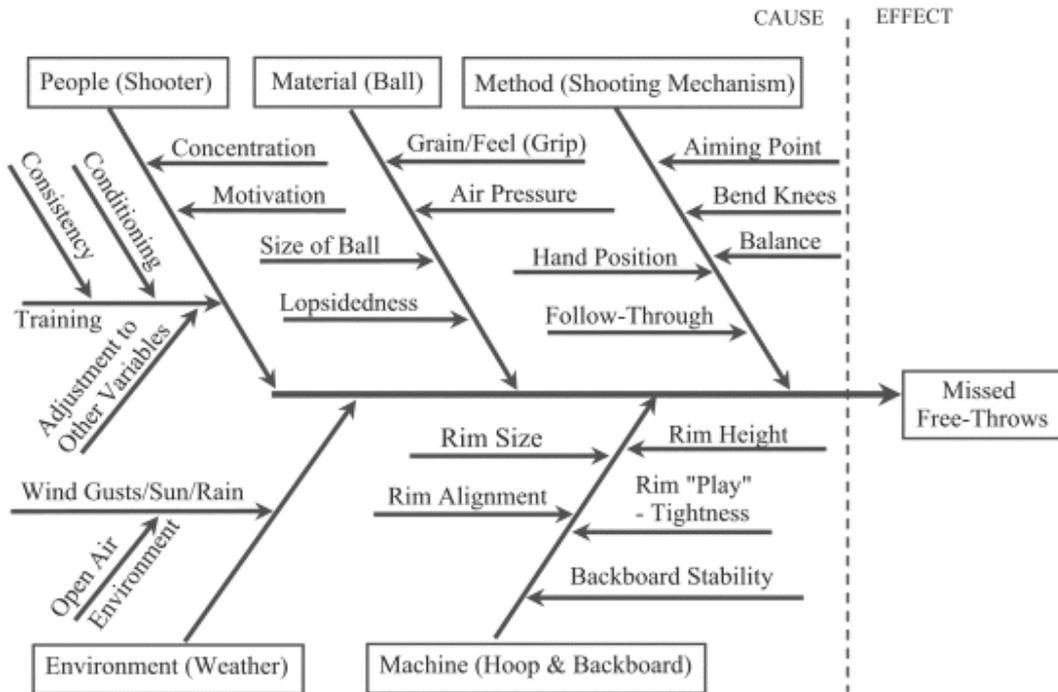


Figure 1.1: An Ishikawa fishbone diagram (reproduced with permission from [Mor])

Checkland’s observation suggests that analytical reduction is not appropriate in the context of an analysis of cause and effect, where an initially holistic approach is required.

Rather than analytical reduction, Empirical Modelling is based on the identification of Observables, Dependency and Agency within experience that is initially personal and subjective.

- An *observable* is a characteristic of my environment to which I can attribute an identity. An observation of an observable returns a current value.
- A *dependency* describes how I perceive the act of changing one particular observable to change other observables predictably and indivisibly.
- An *agent* describes my perception of an entity (a cluster of observables) that is capable of initiating state change. I attribute all changes to the values of observables to agents.

(Beynon *et al* [BCSW99] gives an explanation of these concepts, as understood

1.1. Dependency in concept: EM, Radical Empiricism and the making of meaning

within a system domain. It gives a more ‘operational’ explanation of the concepts, which I will return to in §5.1.1.)

Empirical Modelling is the name we have given to the activity of building artefacts to embody patterns of observables, dependencies and agent actions that are encountered in experience. EM involves the progressive development of understanding through interaction, whereby meaning is continually refined in the light of additional experience.

The use of artefacts for development of understanding is particularly relevant to the experimental activity that precedes the formulation of a theory. In this respect, it is related to the concept of ‘construal’ introduced by Gooding in his philosophical analysis of experimental procedures in science [Goo01]:

...I have labelled interpretative images and their associated linguistic framework as ‘construals’. This term denotes proto-interpretative representations which combine images and words as provisional or tentative interpretations of novel experience.

The most primitive understanding of state change is based on pure agency. For instance, in the absence of alternative explanations, in some situations, we are prone to regard change as stemming from autonomous action. This is illustrated in the way in which the Elizabethans attributed change to agents with different levels of privilege to enact state change within the Chain of Being, beginning with God and passing to the earthworm via planets, royal personages and peasants [Til43]. Dependency is associated with a recognition that one change *entails* another. As we start to perceive dependencies within experience, we move from the perception of independent change that is represented in its most extreme form in animism⁴ towards a view that presumes more predictability and interconnection.

The personal pronoun is used advisedly in the definitions of observable, dependency and agent above. Characterisation of experience in terms of observables, dependency and agency is initially a private, personal matter as it is based on one’s own experience. EM follows Radical Empiricism in taking private experience as primary. Regions of stability are rare within the totality of experience, and so only a relatively small part of experience is associated with stable public ‘theoretical’ knowledge. The learning activities associated with the transition from pre-verbal

⁴The term animism denotes the belief that a soul or spirit exists in every object, even if it is inanimate. It was controversially linked with religion by E.B. Tylor in 1871.

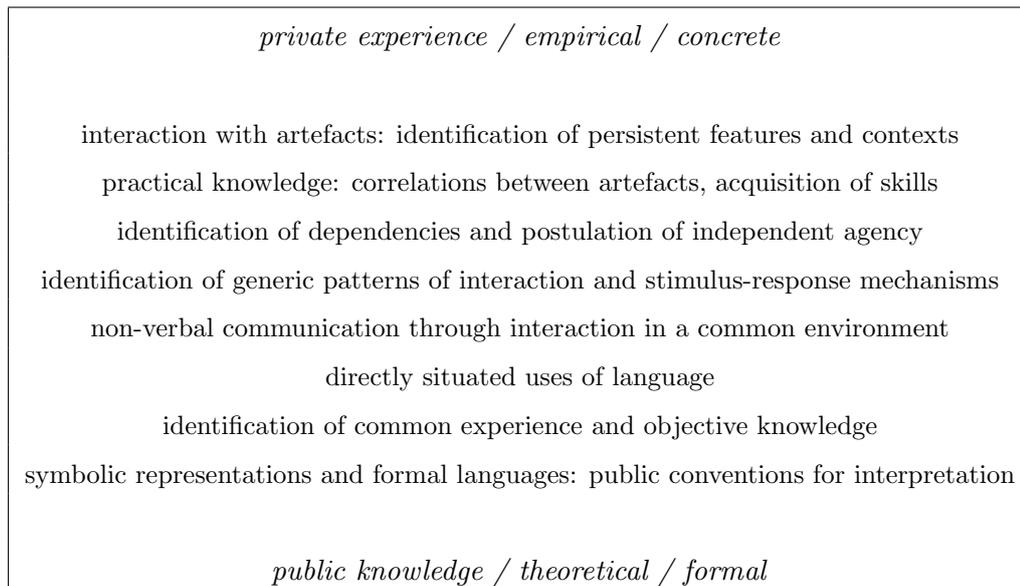


Figure 1.2: The Empiricist Framework for Learning (from [Roe03, p.74])

private experience to public knowledge are set out in the Empiricist Framework for Learning (Figure 1.2) [Roe03].

We use empirical investigation, then, in order firstly to gain private understanding that we may then attempt to make public. The development of understanding (private or public) may be our sole aim, or we may wish to *use* the understanding to make predictions — in particular, to examine the potential consequences of a particular action.

But many experimental environments are ‘noisy’ — there are many agents operating and it is difficult to distinguish the effects that arise indivisibly from one’s own actions from those that arise from the actions of other agents [BNR95]. In some environments (for example, in the train accident scenario), experimental interaction may be dangerous, or the time between stimulus and response may be excessively long, or the values of observables may be hard to determine. These are some motivations for the use of modelling. We construct an environment to model the *referent*, and take measures to make the modelling environment ‘quieter’, safer, swifter in response and more perceptible⁵ in observation than the referent, as appropriate.

⁵In this chapter, I sometimes use the word perceptible to qualify the word observable, meaning that the modeller can determine its value *directly*, rather than through some inference, in the spirit of Radical Empiricism.

1.1. Dependency in concept: EM, Radical Empiricism and the making of meaning

A ‘quiet’ environment is one in which the modeller is the sole agent, providing a tractable context in which all changes to values of observables are due (directly or indirectly) to the modeller’s interaction. We refer to a model of such an environment as a *1-agent model*.

The machine initially developed by Bill Phillips in 1949–50 to demonstrate the circular flow of income of the open-economy IS-LM model [Bar00, p.103] is an example of a 1-agent model [Phi00, p.68]:

Fundamentally, the problem is to design and build a machine the operations of which can be described by a particular system of equations which it may be found useful to set up as the hypotheses of a mathematical model, in other words, a calculating machine for solving differential equations. Since, however, the machines are intended for exposition rather than accurate calculation, a second requirement is that the whole of the operations should be clearly visible and comprehensible to an onlooker.

The Phillips machine used the flow of water through transparent pipes and the gathering of water in reservoirs as a metaphor for the flow of income (see Figure 1.3). As the quote describes, the hydraulic solution was chosen for pedagogical reasons. Water flowing in transparent pipes made the observables perceptible — the possible alternative, electronic computers, at that time had no visual display units. The Phillips machine was a 1-agent model (leaving aside the pump that forces water to the top of the machine so it may cascade down) that was safe (leaving aside the possibility of leakage) and allowed experiments to be run faster than real time.

On what basis does the flow of water in the Phillips machine model represent the flow of income? Water and income are experiences of a very different kind. The experience of the water is said to represent the experience of income in respect of the modeller’s pragmatic conventions for interpretation. This concept once again has a precedent in the work of William James, who discussed “the ways in which one experience can function as the knower of another”⁶. If, in the view of the modeller, experience of the model relates to experience of the referent, then the modeller may regard the model as *meaningful*.

EM supplies environments similar to that provided by the Phillips machine, where meaning can be negotiated through interaction [BS99] with the model, inter-related with other situational, explicit, mental and internal (SEMI) aspects of state

⁶This aspect of James’s work is related to EM in [BS99].



Figure 1.3: The Phillips machine in the London Science Museum, with the author shown for scale. Note that the machine has been drained of water for preservation purposes. Photo: Mark Lloyd.

in the mind of the modeller [BRWW01]. The meaning comes about through the confluence of modeller, the model and these aspects of state — it is not a property of modeller or model alone. This, then, is our interpretation of the word ‘meaningful’. It may seem a rather ‘informal’ definition, but our intention is to circumvent the difficulty of formalising personal understanding by grounding meaning in direct experience, in the manner of the Radical Empiricism of William James.

1.2 Dependency in application: Interaction with Meaningful State

This section briefly reviews digital computer artefacts that exploit dependency similar to that represented in Empirical Modelling to enable *interaction with meaningful state*.

Spreadsheet programs⁷ are the most common example of an experimental environment similar to the Phillips machine, implemented on a digital computer. If a spreadsheet model of aspects of Keynesian economics is constructed (see below), the resulting spreadsheet provides a 1-agent environment in which experiments with the modelled economy can be performed safely and faster than real time. At all times, the values of spreadsheet cells are perceptible. The cell values however require more interpretation on the part of the modeller to form meaning than is required with the Phillips machine. The raw digital numeric values displayed by a spreadsheet do not provide the direct perception of absolute values and rate of change that the analogue display of water levels in the Phillips machine gives⁸.

A spreadsheet program provides an environment that also enables interaction beyond simple use. A spreadsheet is constructed incrementally through progressive interaction. At any stage, ‘what-if?’ interactive experiments can be performed on the partially constructed spreadsheet in order to determine its response to a particular stimulus. If the response of the spreadsheet corresponds to that experienced

⁷Of which the first example was the 1979 [CKA96, p.251] VisiCalc by Bricklin and Frankston [BF].

⁸This deficiency can be rectified somewhat through the use of graphing facilities in the spreadsheet — but current spreadsheet programs do not provide the facility to create a graph as interaction proceeds over time.

with the referent, then the spreadsheet will be considered to have meaning⁹. The spreadsheet can always be extended or modified by adding or changing formulae, so it may never be considered ‘complete’.

These properties of the spreadsheet environment that enable interaction for construction as well as use are also to be found in the Phillips machine, but construction and modification of the Phillips machine would require significantly more specialised engineering knowledge than that required to use the machine. In contrast, a spreadsheet may be constructed or modified, using interaction techniques that are the same (in many cases) as those used for spreadsheet use.

Database environments are another common example of software that attempts to model a referent — perhaps the reservation status of seats on a particular planned future aeroplane flight. Much effort has been expended in attempts to formalise the meaning of data held in databases in an objective way, independent of direct experience. The major problems such attempts encounter are illustrated in Kent’s *Data and Reality* [Ken78], which contains many examples to support his hypothesis that formal modelling is inadequate:

... there is probably no adequate formal modelling system. Information in its ‘real’ essence is probably too amorphous, too ambiguous, too subjective, too slippery and elusive, to ever be pinned down precisely by the objective and deterministic processes embodied in a computer.

Both spreadsheets and databases emphasise interaction with meaningful *state* rather than automation of *transitions*. More examples of other computer applications with the same emphasis include word processors and music creation packages.

Ten principles to guide the designer in implementing interaction with meaningful state are stated in [app87]; they include recommendations to use metaphor, provide stability, direct manipulation and feedback. For example, the original Apple Desktop Interface is a *metaphor* for an office desktop. The desktop is a surface where users can keep documents which are perceived as *stable* — i.e. this is a 1-agent environment. The environment can be *directly manipulated* with *immediate feedback*.

Such interfaces give the perception of interaction with stable *artefacts*. It is the stability of values of observables as determined through interaction that gives artefacts coherence and meaning. In the Phillips machine, the fact that the total

⁹Note: both the correspondence and meaning are initially in the view of the modeller, as before.

amount of water in the machine does not change (modulo leaks or additions) is one of the guarantees of stability that enhances the machine’s meaning. Such guarantees of stability in digital computer models must be programmed. Our current methods of programming make it easy for these guarantees to be omitted or subverted. For example, a currently recurring topic of much interest in the “Forum on Risks to the Public in the Use of Computers and Related Systems” (RISKS-LIST) [Neu] is the use of electronic voting machines without voter verification facilities. The problem occurs due to the lack of guarantees that votes that are made are correctly recorded and counted. Votes are entered into a black-box computer system and their consequences later emerge. The internal states are not perceptible to voters, unlike paper ballots or the water in the Phillips machine.

Interactive Situation Models (ISMs) [Sun99] are the name we have given to our computer artefacts that are intended to support Empirical Modelling. The artefacts are constructed with computer tools that give explicit support for reliable relationships, which can produce perceived stability.

Charles Care’s model of a planimeter is a recently produced example of an ISM. The planimeter is a physical instrument, mostly commonly used around 1850 for land surveying — see Figure 1.4. In the type shown in the figure, a small wheel is placed in contact with a large disc. When the disc is rotated, friction causes the wheel to rotate in response. The wheel is positioned over the disc using a cantilever. The wheel can be moved horizontally across the disc (slipping as it does so) in order to change the gear ratio of disc to wheel movement. The movement of the disc and cantilever can be controlled by the horizontal and vertical motion of a pointer, shown in the foreground of the picture. Typically the pointer would be traced around an irregular closed curve, such as the boundary of a parcel of land on a map [ABCK⁺90, p.167]. The wheel movement is then related to the traced area.

Care’s planimeter is implemented on a digital computer — see Figure 1.5. The mouse pointer can be moved within the square table shown at the top left of the figure. Movement in one axis causes the large disc to rotate, in turn causing movement of the small wheel. Movement in the other axis causes the wheel to move horizontally across the disc, slipping as it is moved¹⁰.

¹⁰The Sasami feature used to produce the 3-dimensional display shown on the right of the figure is further discussed in §4.2.1.

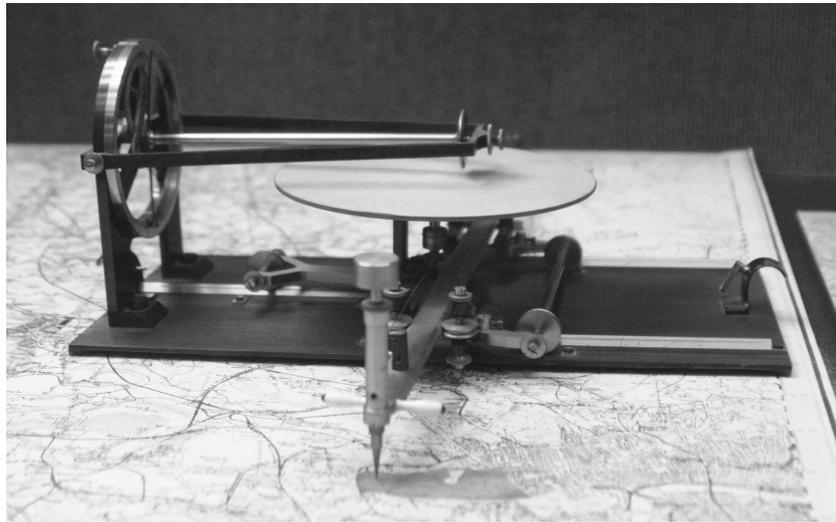


Figure 1.4: A physical planimeter shown in the London Science Museum (photo: Mark Lloyd)

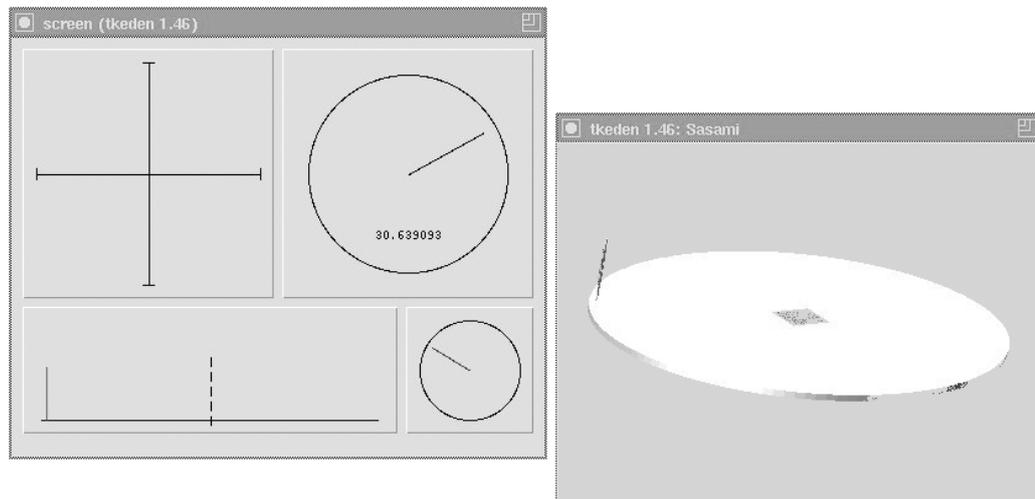


Figure 1.5: Charles Care's Planimeter ISM

Care's computer artefact represents the physical counterpart and has meaning if the user understands the conventions used by Care and stays within the realm of experience modelled by Care (described roughly above). An example of an aspect of experience that is not currently included in the ISM is as follows. The connection between the rotation of the disc and that of the wheel is maintained imperfectly by friction in the physical artefact and slippage occurs in some circumstances. Care's ISM does not at present model slippage. The relationship between disc and wheel is maintained perfectly by the ISM (floating-point imprecision aside), much as relationships between spreadsheet cells are maintained by the spreadsheet program, as I describe later in this section.

Our aim in developing an ISM is initially to produce a computer *instrument*, rather than a computer *tool*. This emphasis reflects the desire to create an artefact that enables interactive observation, together with the need for interpretation. Beynon *et al* [BCH⁺01] give more details, in particular characterising instruments as “maintaining relationships between state”, echoing James's concept of an experience that ‘knows’ another, mentioned earlier (p.8).

The computer-as-instrument theme is developed further in [BWM⁺00], which describes the development and use of a prototype ‘timetabling instrument’ ISM. This ISM (*projecttimetableKeen2000*) is shown in Figure 1.6. It was used to assist with the timetabling of 120 student presentations in 2000 and 2001. The main aim of the model is to provide the timetabler with perceptible state as a basis on which to make decisions — this taking precedence over provision of automated scheduling routines. The model highlights double-booked rooms, clashes between bookings and availabilities, and other problems.

The timetabling instrument ISM illustrates a distinction between perceptible and meaningful state. The timetabling of two student presentations simultaneously in one room could be considered meaningful state: this is a situation that can be imagined and may be necessary or desirable in some circumstances. The timetabling of one person to two rooms simultaneously however is merely a perceptible state of the ISM: existence of the same person in two places simultaneously is not meaningful state for most modellers.

A third ISM, describing a digital watch interface, is shown in [BCSW99]. The

1.2. Dependency in application: Interaction with Meaningful State

screen (dikeden 1.40 (server))

Staff	ML	JES	MCK	SC	SER	AM	SB	SGM	GRN	JB	Rooms	101	104	Message	Welcome to the Temposcope
	OW	VMB	CS	NS	NR	JS	AT	LAG	DA	SAJ	313	211	Board		
	PWG	RGW	TJA	AS	NG	MSP	MSJ	FN	THEC	NAP					
	MAR	RL	GRM	SCMDM											
Slots	9:00-9:40 9:40-10:20 10:20-11:00 11:00-11:40 11:40-12:20 12:20-1:00 1:00-1:40 1:40-2:20 2:20-3:00 3:00-3:40 3:40-4:20 4:20-5:00 5:00-5:40														
Mon Slots 1-13	Helser 104 Skpper 313 Seragos 101 Webb Ma 101 Dean Si 104 Siu Kev 104 Damani 101 Barton 313 Tang Ti 313 Martin 313 D'Orazi 104 Hudspit 104 Arnold 101 Cheng K 101 Li Tsan 313 Earl Le 104 Tanner 313 Gates S 101 Uttaach 211 Twigg A 101 Doorga 101 Avent C 101 Bowen A 313 Kouliko 211 Aziz Ar 104 Bridgev 313 Khuntci 211 Angeli 313 Brose F 211 Bovatt 211														
Tue Slots 14-26	Howell 313 Lee Chu 101 Brooks 104 Ling Ti 104 Sarkar 104 Godinho 104 Jones I 104 Todd Jo 101 Starlin 313 Witham 313 Fewkes 101 Sehra H 313 Shepher 313 Haines 101 Pallot 313														
Wed Slots 27-29															
Thur Slots 40-52	Burr Da 104 Mereuta 104 Gordon 313 Sidney 104 Terbraa 313 McLella 313 Kamolvi 313 Turner 104 Ryder G 104 Hartley 313 Reddish 313 Grainge 313 Genson 313														
Fri Slots 53-65	Protoge 104 Taylor 101 Gudka R 104 Taylor 313 Lee Lin 313 Milward 104 Dowell 313 Pavelin 104 Stewart 211														
Functions	Angeli Andrew Arnold Warren Aziz Arifil Brideswater Cheng Kar leonD'Orazi-FlavonDamani Shivani Dooroa Robin A Gordon Neil A Gudka Rina M Haines Jacobi Liot Khuntci Jasal Koulikov Lee Lincoln KCLing Ti Milward David Pallot Jessica Pavelin Joanna Protooerellis Ramanathan Sarkar Aseeh Seragos Sidney Ian J Siu Hon Tina Terbraak Marc Uttachandani Yung Young Ea Atkinson Avent Barton Oliver Bowen Adam MA Bovatt Russell Brooks Steven Brose Ference Carrott Neil G Carter Thomas Chaloner Chev Rachel Ciccan Dana I Collier Stuart Corforth Cullen Ryan T Dean Simon Dowell James Dumont Earl Lee J Fenwick Andrew Fewkes Adam Forster Henry Front Aviad Gankle David Gates Stephen Genson Jason Kirsinger Ian Chansen Samuel Hartley He Le Ting Hertill Zoe L Howell Stephen Hudsoth Paul Jimenez Coelho Jones Ian M Kamolviit King Samuel J Koszerek Lee Chuin Yao Lee Ling Lun Leonard Carl M Li Tsan Man License Dean Lund George A Mahmood Nassar Martin John D Mason Barry JSMcLellan Mereuta Vlad O Modhvia Amit Morris Graham Parkin Payne Shaun JP Pooe Richard L Ralman Timothy Reddish Reid Matthew A Ryder Greg DO Sehra Harrit Senior Joe T Shepher d Emma Siska David Skeoper Southate Staines Jacob Starling Stewart-Smith Sweetman Tang Timothy Tanner Darren Taylor Taylor Sandip Thomas Stephen Todd John JR Triplov Peter Turner Steven Twigg Andrew Virdi Inderjit Ward John Ward Thomas M Whitewright Witham Duncan Wong Showru Young Robert D Webb Matthew Burr Dale Suarez Gary Godinho Emille Ahi-Rhan Box Malcolm Gordon Kevin Hyde Simon C Kurdi Ahmad Rudnicki Siu Kevin K Y Woodrow Helser														

Figure 1.6: The Temposcope ISM

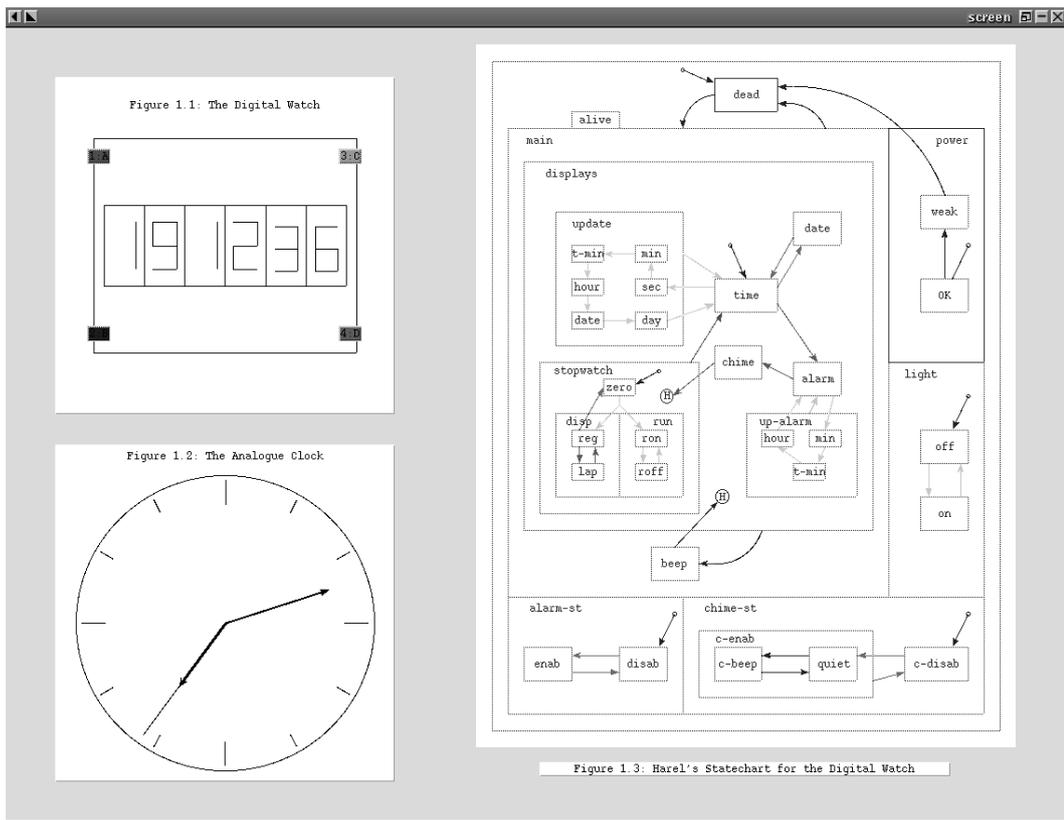


Figure 1.7: The digital watch ISM

paper attempts to show that ISMs can be used to provide support for the design of information systems. In this case, it could be imagined that an embedded system is being designed. The ISM includes a statechart [Har87] in order to make the internal state of the watch artefact perceptible to the user — see Figure 1.7. The paper gives an LSD account of the digital watch, expressing how the original modeller conceived the operation of the watch in terms of observables (states, oracles and handles), dependencies (derivates) and agents. We return to LSD accounts in §2.1.1.

The ISMs above are each implemented using the same tool which is named EDEN. As has already been mentioned, I have been the principal developer of EDEN since October 1999. In terms of enabling Empirical Modelling, EDEN is our most successful tool so far. This thesis examines EDEN closely in Chapter 4 and also two other tools: **am** (an implementation of the ADM, in Chapter 2) and the DAM machine in Chapter 3. EDEN is more focussed on interaction with the modeller than the other two tools.

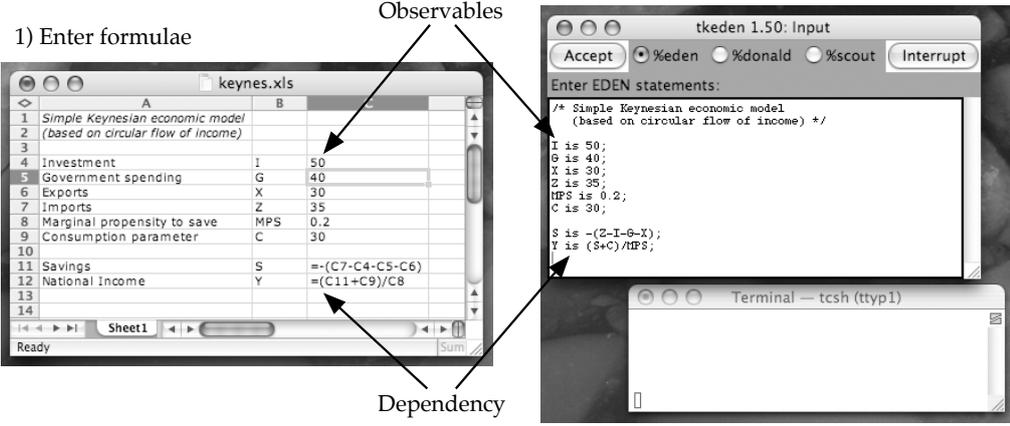
EDEN implements interaction in terms of observables, dependency and agency. Interaction with EDEN, if it is used in a principled way, is more characteristic of interaction with a spreadsheet than with a conventional programming environment. Each of the EDEN-based ISMs described above, similar to a spreadsheet, 1) is provisional and incomplete; 2) can be modified at any time for revisions or ‘what-if’ experimentation; 3) is constructed incrementally. As in the spreadsheet, use and modification of models in EDEN can entail the same kind of interaction.

Figure 1.8 compares use of a spreadsheet program and EDEN when modelling a situation simplified from that represented by the Phillips machine. Use of the concepts of observable, dependency and agency is highlighted in each.

The top right of Figure 1.8 shows a small set of definitions written in the Eden language. A *definition* represents an indivisible relationship between observables by associating a single expression with each named observable. Similar to the spreadsheet formula, a definition implies a one-way relationship: changes to the values of observables on the right-hand side imply a change to the value of the observable on the left-hand side, but not *vice versa*. Taken together, a set of definitions forms a *definitive script*. We term a change to a definition a *redefinition*. The act of making a definition or redefinition is an act of *agency*.

1.2. Dependency in application: Interaction with Meaningful State

1) Enter formulae



Observables

Dependency

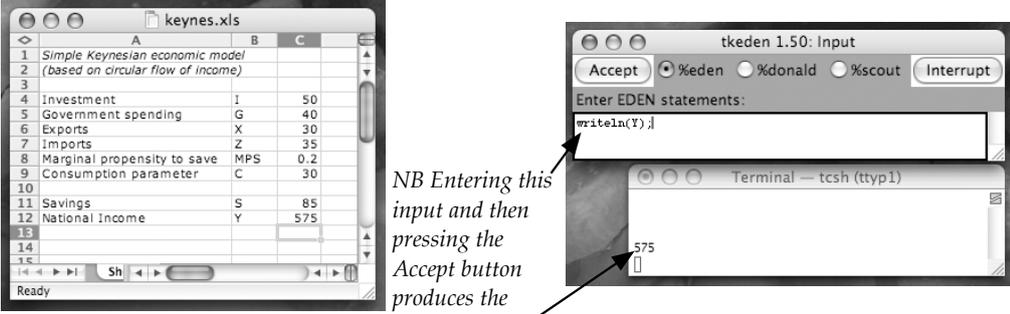
tkeden 1.50: Input

```

Accept %eden %donald %scout Interrupt
Enter EDEN statements:
/* Simple Keynesian economic model
(based on circular flow of income) */
I is 50;
G is 40;
X is 30;
Z is 35;
MPS is 0.2;
C is 30;
S is -(Z-I-G-X);
Y is (S+C)/MPS;

```

2) View initial state



NB Entering this input and then pressing the Accept button produces the result shown here

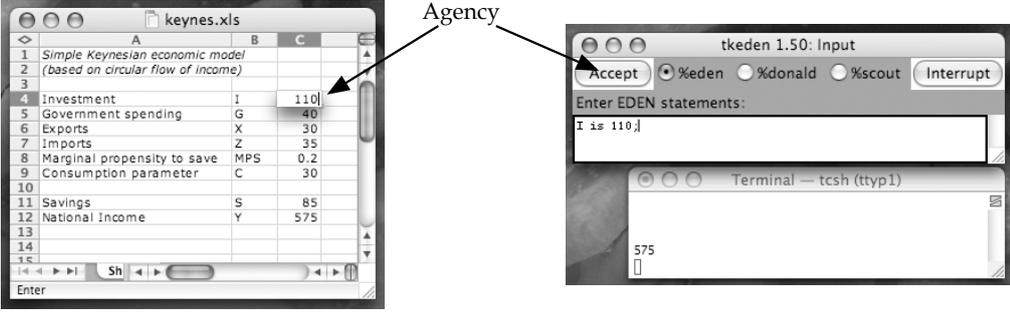
tkeden 1.50: Input

```

Accept %eden %donald %scout Interrupt
Enter EDEN statements:
writeln(Y);

```

3) Change a value (or formula)



Agency

tkeden 1.50: Input

```

Accept %eden %donald %scout Interrupt
Enter EDEN statements:
I is 110;

```

4) View new state



tkeden 1.50: Input

```

Accept %eden %donald %scout Interrupt
Enter EDEN statements:
writeln(Y);

```

Figure 1.8: A spreadsheet program and EDEN, showing observables, dependency and agency

The dependency in a definitive script forms a graph. Appendix §3.A (p.178) explains the *script graph*. Cycles are not permitted in the script graph. (Note that the Phillips machine describes a cyclic process, which we cannot model directly in a definitive script.)

A definitive script is written using syntax specified by a particular definitive notation. We have created many domain-specific *definitive notations*, each providing operators and types specific to the domain.

There are now many sources that examine EM and EM in application. Along with 80 refereed journal and conference papers, there are currently 25 research reports, eleven PhD theses, five MSc-by-research theses, ten taught MSc project reports (including one by myself [War98]) and numerous third year project reports.

Existing documentation was not a problem to the ‘first generation’ researchers of EM, but the now abundant collection poses a significant challenge to a researcher working on the ‘second generation’ of EM today. In various attempts to improve the organisation of our library, I have created an EDDI database of many of the paper documents (the EDDI database environment in which this particular database is constructed is described in §4.2.2 and [BBRW03]); organised the scanning of much material; and maintained reference lists for our website. I have also designed and implemented a web archive ‘empublic’ [WRB] to hold our electronic data from various projects (mostly ISMs), dating from 1987. With the assistance of Chris Roe and Meurig Beynon, the archive now holds over 120 projects.

EM has broad scope of application, as can be seen by looking at just the subjects of the eleven PhD theses. The classification of ten of these theses by subject area gives an idea of the extent of potential application.

- Computer Aided Design / Geometric modelling — [Car94], [Car99]
- Business — [Ras01], [Ch’01], [Maa02]
- Educational technology — [Roe03]
- Software system development — [Yun93], [Nes97], [Sun99], [Won03]

This section has tried to give a flavour of the various applications of EM to modelling. More information can be found in the PhD thesis that has been omitted

from the above classification:

- A Treatise on Modelling with Definitive Scripts — [Run02]

which — rather than focussing on a particular application — gives an overview of modelling with definitive scripts.

This thesis is complementary to [Run02] in that, rather than focussing on a particular application, it concentrates on the problem of *implementing* dependency.

1.3 Dependency in development: a novel abstraction

Conventionally, in Computer Science we make a sharp distinction between software use and software development, corresponding to the human roles of user and programmer and to the distinction between program and data in the computer. In the interests of clarity, it is helpful in this introduction to make the conventional distinction between *application* and *development*.

However, in the previous section, we indicated that use and modification of a spreadsheet or ISM often entails the same kind of interaction. Changing the contents of a spreadsheet cell or making a redefinition can be construed as a use or a modification of the artefact. Often, one such interaction can be construed in either way.

Dependency is also an abstraction that (unusually) conflates program and data. For example, the definition ‘*a is b+c*’ denotes a data variable ‘*a*’ and also describes a recipe for recalculating its value.

In Empirical Modelling, the distinction between application and development is then a rather artificial one. Empirical Modelling is not only an activity to be implemented, for which we seek a supporting computer tool. The concepts of observable, dependency and agency and the underlying principles of EM¹¹ are also applicable to software development itself.

A full exploration of the concept of dependency in software development is outside the scope of this thesis, where the primary focus is on implementation. Many

¹¹The expression “EM concepts” refers to the concepts of observable, dependency and agency. The distinct expression “EM *principles*” usually refers to the *application* of those concepts.

other sources (including the four PhD theses listed under ‘Software system development’ in the previous section) are concerned with this exploration. These theses and the body of practice of the EM research group since its inception in 1981 (some of which is recorded in the ‘empublic’ archive [WRB]) indicate that dependency provides a kind of abstraction which:

- is novel;
- is flexible;
- provides useful guarantees;
- is so simple as a concept that it is embodied in the use of the keyword ‘is’ in `a is b+c`, and
- may be implemented in a variety of ways, some of which may be quite simple.

This section indicates ways in which these qualities of dependency as a novel abstraction are significant in software development.

The construction of software systems is still a hard problem, despite over fifty years of accumulated experience. Neumann [Neu95, p.309] presents a table summarising cases of computer-related risk described in the RISKSLIST¹² archive over the period 1985–1993. It shows a total of 1174 cases, 81 of which have each involved at least one death. A recent article [GMT04] reports six fatal accidents since 1993 involving software-related problems.

A (highly simplistic) division of the problems in constructing software systems is:

P1 How can we know what we want?

(The requirements problem)

P2 How can we know we have constructed what we want?

(The comprehension/validation problem)

Frederick Brooks [Bro87] divides the difficulties in software construction into accident and essence, and states that the accidental problems are largely solved.

¹²Forum on Risks to the Public in the Use of Computers and Related Systems.

He considers the “irreducible essence of modern software systems” to have four inherent properties: conformity, changeability, complexity and invisibility. Brooks’s conformity and changeability correspond roughly to problem P1, complexity and invisibility to problem P2. The following subsections briefly examine problems P1 and P2 respectively.

1.3.1 Modelling requirements

The full difficulties of problem P1 were not at first recognised, perhaps because of the initial prevalence of ‘accidental’ problems. In ‘The Mythical Man-Month’ [FPB95], originally written in 1975, Brooks recommends that we “plan to throw one away; you will, anyhow”. Brooks justified discarding the prototype partly due to the evolution of the requirement whilst learning happens during construction. In [Bro87], he goes further, stating that “The hardest single part of building a software system is deciding precisely what to build.” In a review chapter added to [FPB95] in 1995, Brooks recommends the ‘incremental-build model’ over the waterfall model, ‘growing’ the software and having a running system at every stage. This approach allows the modelling of requirements. Many other authors concur — although there is some controversy over whether requirements should be modelled before or during development.

- The construction of ‘use-cases’, recommended as a starting point for object-oriented design [JC92], is an explicit attempt to model requirements to guide the development that follows.
- eXtreme Programming (XP) methods [Bec99] include the recommendation to keep the system as flexible as possible so that development can proceed in any direction as the requirement is realised. An XP project starts with a quick requirements analysis which continues throughout development.
- [Coo99] asserts that “all programming is design” which affects the possible interaction with the final product. He recommends primacy of ‘interaction design’ and emphasises that this must come *before* construction.
- Open Source as a methodology is partially justified on the basis that programmers will “scratch their itch” [Ray]. Open Source software can in principle

be modified to the personal requirements of the user, if that user is also a programmer.

Modelling assists by giving meaning to requirements — it is easier to evaluate an artefact that can be directly perceived and manipulated than an abstract requirements specification. The potential application of EM to requirements modelling has been a prominent theme in much previous research [BR95, Sun99, BCSW99, Ch’01] and will not be considered further in this thesis.

1.3.2 Program comprehension and validation through abstraction

Problem P2 (“How can we know we have constructed what we want?”) corresponds roughly to the remaining two of Brooks’s “inherent properties of [software’s] irreducible essence”: complexity and invisibility [Bro87].

The Unified Modeling Language (UML) [BRJ99] specifies a collection of (currently twelve) types of diagram for visualising some aspects of software. [Mil02] describes the aim of the Unified Modeling Language (UML) as:

... the kind of tool mature engineering disciplines have had for centuries — a shared graphical language for descriptions and specifications.

He then introduces the debate surrounding the UML specification, concluding that:

We’ll know we have what we need when, as with blueprints, topographic maps, and circuit diagrams, such debates are no longer necessary.

Following Brooks, who states that “the reality of software is not inherently embedded in space” [Bro87], I would argue that software as currently constructed does not lend itself to uncontroversial diagrammatic representational forms.

In contrast, software constructed using dependency *can* be represented in an uncontroversial diagrammatic form, making use of “script graphs” (to be described in Appendix §3.A, p.178) that are directly analogous to circuit diagrams.

How can complexity be tackled? It is still a problem of essence. Dijkstra [Dij01] echoes Brooks, making a distinction between intrinsic and accidental complexity:

I would therefore like to posit that computing’s central challenge, “How not to make a mess of it,” has *not* been met ...

...

... while we all know that unmastered complexity is at the root of the misery, we do not know what degree of simplicity can be obtained, nor to what extent the intrinsic complexity of the whole design has to show up in the interfaces.

We simply do not know yet the limits of disentanglement. We do not know yet whether intrinsic intricacy can be distinguished from accidental intricacy. We do not know yet whether trade-offs will be possible. We do not know yet whether we can invent a meaningful concept for intricacy about which we can prove theorems that help ...

The conventional way to cope with complexity is to use abstraction. Abstraction can be defined ([LT77], quoted in [Bis86]) as

a general idea which concentrates on the essential qualities of something, rather than on concrete realizations or actual instances.

Abstraction therefore relies on the principle of analytical reduction as defined earlier (p.4) and is subject to the limitations stated there.

The philosophy of Radical Empiricism that underlies EM seems at first to preclude abstraction. Earlier, I paraphrased Wild's observation [Wil69] to the effect that: we may have our own systems and conclusions, but must be ready to examine any new fact at any time and make the necessary revisions and corrections. This suggests that in using EM for software development, we cannot abstract away any details.

However, the concept of dependency is compatible with Radical Empiricism. Dependency can be considered to be a type of abstraction¹³, but one rather different from the other types of abstraction common in programming. It introduces *relationships* between observables — in the computer, guaranteed, yet reconfigurable, connections within state. Dependency allows revisions and corrections to be made at any time in the light of new experience, but also maintains consistency within state. It provides a way to structure a program in a way that is meaningful to the modeller.

Complexity in experience is not limited to computer programming. I would suggest that making a program *meaningful*, relating it to other complex experience through the guarantees given by computer-maintained dependency, is a promising way to manage complexity.

Dependency is an abstraction that produces meaningful *state*. This differs from

¹³A traditional empiricist would identify dependency as an abstraction whereas Radical Empiricism treats it as a relationship "given in experience".

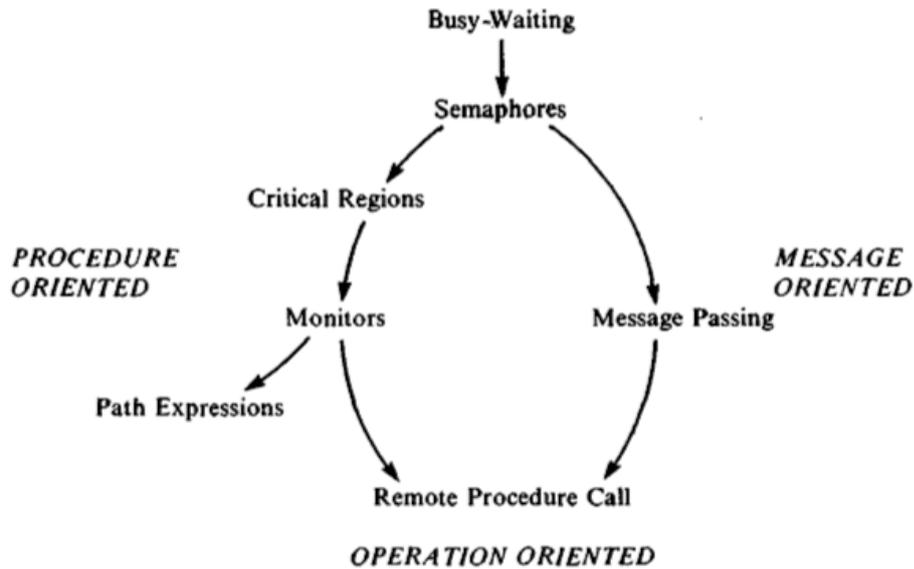


Figure 1.9: Synchronisation techniques and language classes (from [AS83, p.38], © 1983 ACM, reprinted by permission of the Association for Computing Machinery)

other common types of computing abstraction that aim to provide meaningful *operations*. The subroutine is one example of this type of abstraction. The abstract data type (ADT), now incorporated into object-orientation, hides data representation behind a set of applicable operations.

Abstractions used in concurrent programming have a similar focus. Figure 1.9 is taken from [AS83, p.38], where it illustrates historical and conceptual relationships among synchronisation primitives. Brinch Hansen’s book [Han02b], which is subtitled “From Semaphores to Remote Procedure Calls”, also traces this progression.

The abstractions that are currently common in both sequential and concurrent programming are thus aimed at providing meaningful operations. Programs constructed with these abstractions are built from hierarchically abstracted sequences of actions. With reference to EM concepts, the abstractions therefore relate to *agency* rather than dependency and observables.

Such abstractions assist somewhat with program comprehension, since they reduce the amount of operational detail that must be examined. However, I would contend that the meaning of the state (in the terms defined earlier — p.8) is not encoded in programs that use such abstractions. Instead, meaning must be derived

through an understanding of the operations that act upon the state. This is easiest to demonstrate through an analogy.

Sequences of actions are rather like describing a journey to a traveller without interaction with the surroundings. For example, the program code:

```
for (i=1; i<=87; i++) { fwd(); };  
turn(35);  
for (i=1; i<=12; i++) { fwd(); };
```

is analogous to the sequence of directions:

```
“move forward 87 metres;  
turn left 35 degrees;  
move forward 12 metres...”
```

The traveller will reach the destination if the directions were described in enough precision and followed faithfully. If there is a minor error in one step of the directions, the eventual result may be wildly inaccurate. For example, perhaps the second instruction should have read:

```
turn(-35);
```

(*cf.* “turn right 35 degrees.”)

Errors in the directions may be caused:

- by simple slips in construction:
using `++` rather than `--`,
(*cf.* saying ‘left’ instead of ‘right’),
- by a mistaken belief of the constructor:
“I didn’t know that `turn()` set an absolute rather than relative heading”
(*cf.* “I thought we were starting from the post office”),
- by an unnoticed change in the domain:
the dynamic library file containing code for `fwd()` was removed,
(*cf.* the road was closed for repairs).

When sequences of actions are used, there is a short distance from a working to a non-working program. The sequence of actions does not itself contain any clues to the meaning and hence to the locations of any errors: it must be stepped through from the start and the meaning of each step (with reference to the requirement) determined. After the problem is located and fixed, the sequence must usually be re-run from the start.

When dependency is used, in every state there is a wealth of possible transitions to other ‘near’ states. The other ‘near’ states are all perceptible (if the appropriate redefinitions are made) and some are meaningful. Like a spreadsheet, the script is always ‘working’ — although some states have no meaning. When a problem is discovered, experimentation can take place starting in the problem state. When meaning is restored, progress can continue from the point reached.

In contrasting conventional programming with the use of dependency within EM in this way, I speak from practical experience in comprehending and maintaining EDEN. Section §4.2 describes some initial attempts I have made to reconstruct EDEN ‘in itself’ in order to increase the internal meaning of the software. The next section describes dependency yet closer to the machine.

1.4 **Dependency in engineering: simple and consistent relationships**

The type of relationships expressed by dependency are a major factor in engineering disciplines other than computing. Leveson [Lev95, p.509] quotes an unpublished essay [McC] “When Reach Exceeds Grasp” which asserts the need for explicable relationships within software and software systems.

[Software] developers have always had to explain relationships within and between their systems. If they can explain those relationships with the simplicity and consistency demanded of other engineering disciplines, they will succeed. If not, it probably means that a dash for novelty has sprinted too far, too fast, and too soon.

At a high level of abstraction (we shall consider a lower level shortly), a digital combinatorial logic circuit embodies a simple and consistent relationship that can be described with Boolean algebra [Boo54]. The circuit shown in Figure 1.10, for example, describes four internal indivisible relationships. Each of the three AND

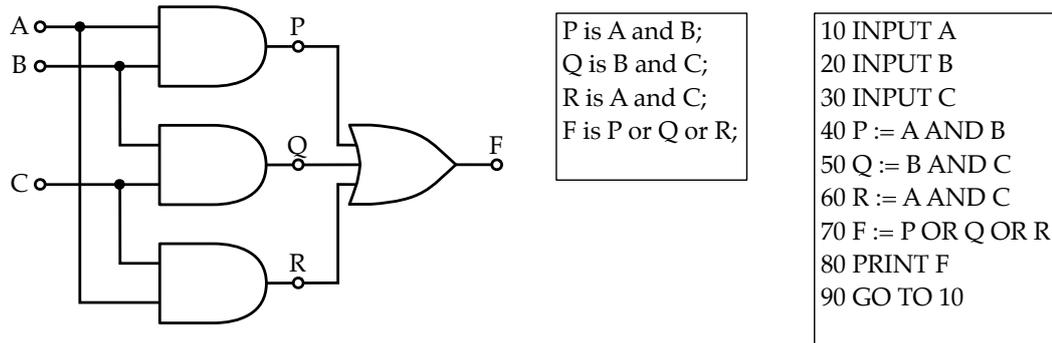


Figure 1.10: A digital combinational circuit, definitive script and ‘i/o equivalent’ computer program

gates indivisibly relates a different pair of values of the inputs A, B and C to the values at points P, Q and R. The OR gate indivisibly relates the values of P, Q and R to that of the output F. These four internal relationships can be summarised to one indivisible input-output relationship. By common convention, this particular interpretation (at this level of abstraction) is intrinsic to how this type of diagram is understood.

In the spirit of this thesis, we could describe the gates as ‘relationship maintainers’. They can then be described with a set of definitions, and the whole circuit forms the definitive script shown in the centre of the figure.

The circuit can be extended or composed with other circuits at any of the named points, with certain known limits: for example, maximum gate ‘fan-out’ limits the number of connections that can be made at any point.

The properties of easy extension and composition are rather like that of a spreadsheet or definitive script. In a spreadsheet, a new formula can be added to an empty and unreferenced cell without affecting the existing spreadsheet. In a definitive script, a new definition can be added to extend the script without affecting the existing script.

The circuit diagram form shown in Figure 1.10 is uncontroversial. A geometric reality is captured in a geometric abstraction [Bro87]. The circuit and circuit diagram are both located in space (which is typically planar) and there is an obvious way to make a mapping between the two. In Appendix §3.A (p.178) I describe the script graph, which is a similarly uncontroversial way of drawing a definitive script.

An input-output relation deemed to be ‘the same’ as that of the circuit can be maintained by a computer program such as the one shown at the right of the figure¹⁴. However, whereas there is a direct correspondence between the definitions in the script and the gates in the circuit in Figure 1.10, there is no equivalent correspondence between the program and the circuit. The intermediate states that arise during program execution (for example, between lines 40 and 50) are not meaningful. In a larger example, it can be difficult to determine when the state is meaningful. Making extensions to the program or composing it with other programs is then a difficult undertaking, compared to extension of the circuit, spreadsheet and definitive script.

Potentially, a single assignment from the program in Figure 1.10 might be seen as corresponding to a single gate in the circuit. For instance, if the assignment

P := A AND B

is wrapped within a procedural program as in Figure 1.11, the input-output relation of the gate in Figure 1.11 and that maintained by the computer program are ‘the same’. However, with the information given, it is not easy to see how the program could be composed with others representing other gates. If the program is able to interact with other programs via the INPUT statements and the assignment, perhaps through shared memory or a data pipe, and if the program can be run concurrently or interleaved with the others, and if the program executes speedily enough, then composition may be possible. Such issues are the theme of this thesis. In particular, the discussion of the operator scheduling of the DAM machine in §3.1.2 and §3.2.5, the operational semantics of EDEN in §4.3 and the concurrent synchronisation in §5.1.2 are directly relevant to these issues. The essential problem is determining how to schedule execution of such gate-programs so as to allow interaction with meaningful state.

It may be noted that the programs in Figures 1.10 and 1.11 are non-terminating and have to be treated in conventional semantic frameworks as mapping sequences of inputs to sequences of outputs over time. The semantic problems of composition of such programs are well-recognised. Relevant references include Milner’s “Elements of interaction” [Mil93], Wegner’s “Why interaction is more powerful than

¹⁴Which is written in a BASIC-like language in order to emphasise the sequencing of actions.

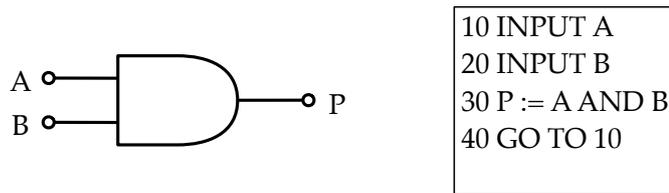


Figure 1.11: A single gate, compared with ‘i/o equivalent’ computer program

algorithms” [Weg97] and Wadge’s proposal for the addition of the ‘hiaton’ as a new type of data object to the Lucid dataflow programming language to enable a nonprocedural dataflow approach to real time [WA85, p.159].

Backus [Bac78] identifies problems in composition and extension as a general feature of “von Neumann languages”. He criticises such languages for a “word-at-a-time style of programming”, in which:

1. semantics are too closely coupled to state transitions: every detail of a computation changes the state;
2. programming is split into an orderly world of expressions and a disorderly world of statements;
3. there is a lack of useful mathematical properties for reasoning about programs, and
4. combining forms cannot be used to build new programs from existing ones.

EM adopts an alternative perspective on computation in which dependency serves to address each of these concerns. Interaction with definitive scripts is not a word-at-a-time style of programming, as a single redefinition may cause the values of many dependent variables — and perhaps the dependency structure itself — to change in response. With reference to (1), Backus’s concern relates to the ‘meaningless’ and low-level nature of transitions in the von Neumann model. In contrast, in EM, the atomic changes of state associated with redefinition match meaningful high-level interactions in the modeller’s mind. Of course, in implementing dependency on von Neumann architectures, it is impossible to eliminate the intermediate states that arise in dependency maintenance. For many practical purposes, this

need not concern the modeller, to the extent that EM tools can provide reliable mechanisms for dependency maintenance. Where the implementation of dependency maintenance is concerned, or where the identification of dependency itself is unusually subtle, a deeper analysis is required¹⁵.

Where (2) is concerned, definitions can be informally viewed as associating expressions with names of observables. A pure definitive script is thus based wholly in the “orderly world of expressions”, and algebraic techniques can be used to manipulate and reason (3) about scripts (although this has not been much explored in our research). Finally — name space clashes aside — definitive scripts can be composed (4) with interesting results.

The functional languages which were highlighted in Backus’s paper (and also the pure versions of the related data flow languages [Hud89, ŠilcRU01]) abstract away variables and hence state. For interaction with meaningful state, this is a major limitation. Backus also considers this a major limitation: “The primary limitation of FP systems is that they are not history sensitive”, and proposes combining an applicative functional programming subsystem with a state and transition rules, forming an Applicative State Transition system (AST system). The definitive systems investigated in this thesis are similar to AST systems in some respects, but more emphasis is placed upon state and state-changing interaction.

So far in this section, digital logic gates have been viewed as implementing an *instantaneous* relationship between inputs and output. The abstraction by which we ignore the time taken for propagation of change allows Boolean algebra to be used.

A similar abstraction is commonly employed when using spreadsheets. We may be aware that the spreadsheet takes time to recompute, but as long as the time is small, we choose to disregard this observation. This abstraction is also a part of the EM research group’s abstraction of dependency. It fits with some of the experience we wish to model — for example, in law, the act of signing a deed by convention instantaneously confers ownership of the corresponding house.

However, in circuitry, the abstraction breaks down in some situations involving particular kinds of observation and particular circuit configurations. For example,

¹⁵*Cf.* the discussion of major and minor transitions in the ADM in §2.3.3.

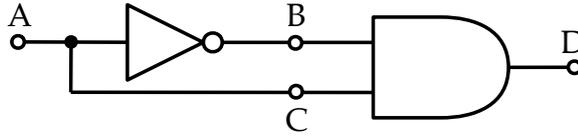


Figure 1.12: A ‘pulse generator’ combinatorial circuit where propagation time is significant (from [Tan99, p.144], © 1999, reprinted by permission of Pearson Education, Inc., Upper Saddle River, NJ)

consider the ‘pulse generator’ circuit of Figure 1.12 (taken from [Tan99, p.144]). If we consider the relationships within the circuit to be maintained instantaneously, then D will always be 0, since B will always be $\neg C$.

However, if the propagation delay through the inverter¹⁶, after a positive edge (0 to 1 transition) occurs at A, the output D becomes 1 for a short period of time. The width of the pulse is equal to the gate delay of the inverter (typically 5 nsec or less [Tan99]).

This issue does not prevent effective use of the higher-level ‘instantaneous’ abstraction. Propagation delay considerations need only be considered under particular observational regimes and with particular circuits. Both these situations are quite simple to specify.

The ‘instantaneous’ abstraction can still be used if observers do not register and/or respond to signals that exhibit this particular timing. The way in which we choose to ignore update time in a spreadsheet illustrates this. In the circuit example, if a human were to observe point D using a simple LED, the flash of light would be imperceptibly short. However, if we were to connect a circuit able to respond to a pulse of the given width to point D, the ‘instantaneous’ abstraction does not apply and propagation delay would need to be considered.

The issue also does not arise if the propagation delay of each path to the inputs of each gate is the same. To achieve this, a commonly used rule of thumb is to construct circuits with the same number of gates in each path. In this example, another inverter could be added in the path between A and C, and the ‘instantaneous’ abstraction then holds.

¹⁶The propagation delay from point A to point C is assumed negligible.

Use of the ‘instantaneous’ abstraction corresponds to treatment of the phenomenon as dependency. If this is not possible, we consider it as agency. However, it is clear that a definition maintainer constructed on a digital computer must implement dependency as agency.

When concurrent definition maintenance is considered, the time for propagation of change becomes significant, in ways similar to that described here. One solution analogous to the solutions commonly used in circuitry is to introduce synchronisation. I examine this topic in §5.1.2. The amount of time taken for propagation of change is also an interesting issue for study in itself, for example because of its potential implications for building reliable software for real-time applications — this is briefly considered in §3.1.2 (particularly p.104).

In this kind of analysis, then, definitive scripts appear to have more in common with circuits than procedural programs. Making the digital computer a reconfigurable analogue device¹⁷ is a theme of this thesis. I include two particular examples that show two ways of considering this. In §3.5.4 I show how the video hardware of the computer can be considered to be performing ‘definition maintenance’. In §4.2.8 I show how the EDEN definition maintainer can be considered to be an extension of a model railway control circuit.

1.5 Related work

This section surveys some of the literature relating to the topic of dependency maintenance and tools that implement it. Dependency is a well-known concept, but due to varied aims and focus, the work is spread across many sub-fields within Computer Science and can use varied terminology. Here, we start by considering application-level tools with the familiar example of the spreadsheet program, proceed through work related to software development and finally close with the dependency concept exhibited in hardware as dataflow computing.

The spreadsheet program provided one of the precedents for the concept of dependency as implemented in EM tools — indeed many EM sources explain the concept through analogy with a spreadsheet (eg [Bey85], [Bey90], [Yun89]). In

¹⁷In the sense that it ‘continuously’ maintains relationships rather than processes real-valued variables.

Figure 1.8 (p.17), we compared a simple model in our EDEN tool with a simple spreadsheet in the Microsoft Excel spreadsheet program.

Campbell-Kelly [CK03] sets out the ‘prehistory’ of spreadsheets. During the 1970s, the word ‘spreadsheet’ referred to a piece of paper ruled with a grid, facilitating the recording of financial data. This changed after the advent in 1979 of an interactive ‘visible calculator’ application for the Apple II computer. It was written by Bricklin, named ‘VisiCalc’ and calculated the effect of changes in (near) real-time. The product was superseded in the market by Lotus 1-2-3 in 1984 and then again by Microsoft Excel in 1990, which dominates today.

Scholarly interest in this type of computing artefact was at first limited. Early papers commonly describe spreadsheet program implementations constructed by the author in a particular language (e.g. in Smalltalk-80 [Pie86] and in a combination of APL and C [Puc87]).

An article by Amsterdam [Ams86] clearly illustrates some issues relating to choices of data structure and algorithms that are relevant to this thesis, by describing a progression of implementations in Modula-2. Amsterdam’s first implementation uses a ‘naïve’ evaluation strategy, re-calculating the value of every cell when any change is made. The second implementation takes the dependency graph established by the spreadsheet formulae into account: a change starts a recursive update of all affected cells. The possibility of a loop in the graph then becomes a problem — the recursive update may become an unbounded loop, following cyclic formula references. Amsterdam suggests fixing the problem by associating a boolean flag with each cell, which is then used to detect the reoccurrence of an already-calculated cell during an update.

A second problem described is one involving indirect forms of reference. (For example, in the modern Excel, the formula ‘=INDIRECT(A1)’ takes the value of the cell referenced by the string currently held in the cell A1.) The recursive update procedure no longer suffices when formulae can use these forms of reference. Amsterdam proposes three ways to resolve the problem. The first proposal does not actually constitute a solution: simply disallow indirect forms of reference. The second involves special treatment of cells whose formula contain an indirect reference. These cells are marked as ‘volatile’. The ‘naïve’ implementation is then used for volatile

cells, re-calculating their value when any change is made. Non-volatile cells can be updated recursively as before. The third, and most complex proposal involves elaborating the dependency graph to include information on indirect references. A requirement then follows to keep the graph information up to date, deleting and adding information when a formula reference changes. I believe that the ‘volatile-cells’ approach is currently used in the Excel implementation. Alternative open-source spreadsheet programs include the OpenOffice/StarOffice ‘Calc’ [Ope, sta], which I believe currently uses the volatile-cells approach, and ‘GNUmeric’ [Gol], which I believe currently uses the ‘elaborate-dependency-graph’ approach.

Amsterdam goes on to address the question of scalability, first eliminating the necessity of storing a representation for every referenceable cell by creating a sparse data structure of ‘chunks’ of cells. He then goes on to describe an implementation where the least-recently-used ‘chunk’ of cells can be swapped out to disk if this becomes necessary.

One might expect the volume of scholarly publications relating to spreadsheets to fall after an initial burst of interest. Figure 1.13 indicates the number of scholarly articles that include the word ‘spreadsheet’ in the title, counted per year since 1979. The two sources used for this search were the ACM Digital Library (a collection of ACM journal and newsletter articles and conference proceedings) [acm] and the IEEE/IEE Xplore service (a collection of IEEE/IEE journals and conference proceedings) [IEE]. The hypothesis that the number of publications would fall seems incorrect, or perhaps the “initial burst” of interest has yet to cease. The count from ACM articles appears to be increasing over time, showing something like a cycle of popularity with a periodicity of 5 years.

Kay [Kay84] distills a large amount of information regarding the spreadsheet paradigm in his *value rule*: a cell’s value is defined solely by the formula explicitly given to it by the user. As formulae are functions, the spreadsheet uses a form of functional language. However, attention from the programming languages community was slow to come. Casimir [Cas92] notes the lack of attention, but then suggests this is because “spreadsheets are intrinsically uninteresting”. He illustrates his contention by attempting to create solutions for traditional program assignments (Fibonacci, factorial, finite automaton, game of life, selection sort, combinations,

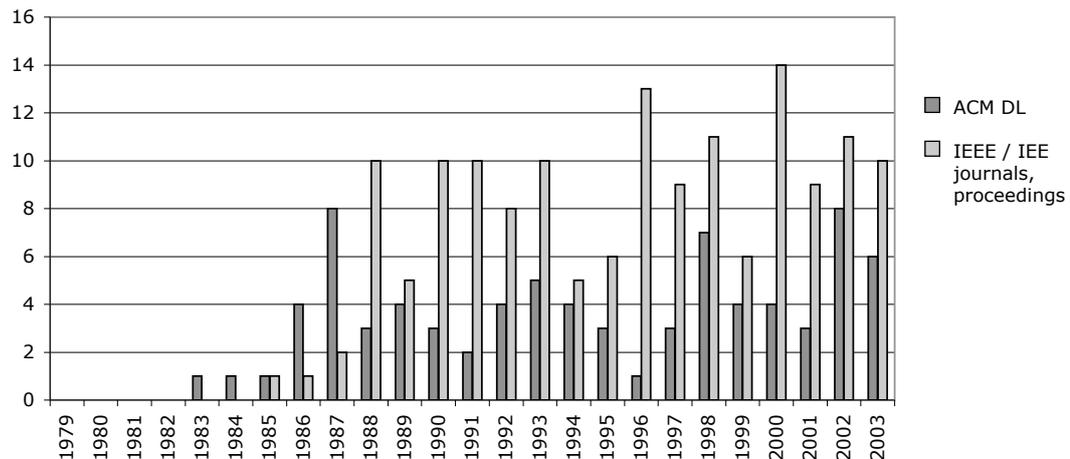


Figure 1.13: Count of articles including the word ‘spreadsheet’ in the title

permutations, towers of Hanoi) in Lotus 1-2-3. He concludes that “...the power of the pure spreadsheet language of Lotus 1-2-3, without the addition of macros, is equivalent to general purpose languages”, but it has various shortcomings, most of which “would have been prevented if the originators of spreadsheet languages had had a more fundamental knowledge of programming languages, especially LISP and APL”. Yoder [YC94] responds to the shortcomings by creating a new spreadsheet language named Mini-SP which is “structured, concurrent and scalable”. It does not appear to have been implemented, although attempts have been made to implement spreadsheet programs in various functional languages (e.g. in Clean [dHRvE95], in Plane Lucid [DW90] and more recently in Haskell [Han02a], [LM02]).

Spreadsheet programs are now very popular. Boehm *et al* [BCHW95] estimates that in 2005, 55 million practitioners will work in the US “end-user programming” sector (using spreadsheets and other “application generators”), compared to only 2.75 million practitioners in the “infrastructure” sector (developing operating systems, database management systems, middleware etc.).

Some research examines how spreadsheets are used in practice in the end-user programming sector. Taking an ethnographic approach, Nardi and Miller [NM90] conclude that spreadsheet programs give strong support to collaborative working of people with different levels of programming and domain expertise. Nardi (an anthropologist specializing in the study of technology) studies end-user computing in “A Small Matter of Programming” [Nar93]. Her findings include the observation

that spreadsheet users can build spreadsheets that fulfill their needs using only a small number of functions in each spreadsheet, normally less than ten [Nar93, p.43]. Most recently, Peyton Jones *et al* [JBB03] propose extensions to the Excel spreadsheet program in order to integrate user-defined functions into the spreadsheet grid, basing their analytical approach on the cognitive dimensions of notations framework [GP96] and the attention investment model of abstraction use [Bla02].

In the late 1980s, articles began to appear stating that erroneous spreadsheet models were an increasing problem. Cragg and King [CK93] summarise the situation and present the results of a survey of spreadsheets, finding that 25% of the models examined contained errors.

Some researchers have suggested ways to reduce the number of spreadsheet errors. Isakowitz *et al* [ISL95] suggest that errors occur “because the lines of logical design and physical implementation are blurred in the conventional setting of a spreadsheet program”. They suggest that every spreadsheet can be characterised by four principal properties:

$$\text{spreadsheet} = \text{schema} + \text{data} + \text{editorial} + \text{binding}$$

which is intended as a contrast to Wirth’s [Wir76]:

$$\text{program} = \text{algorithm} + \text{data structure}$$

They describe a means by which these four components can be extracted from a spreadsheet and then later synthesised into new spreadsheets.

As an alternative, Fisher *et al* [FCR⁺02] describe a “What You See Is What You Test” methodology whereby as a spreadsheet is incrementally developed, it can also be incrementally tested. The user validates calculated values given current input data by ticking “decision boxes”. The spreadsheet program tracks the test coverage over time and through cell dependencies, graphically highlighting untested cells.

The success of the dependency concept employed by the spreadsheet program has encouraged some to build similar applications in order to experiment with the possibilities. Each application adds to the basic spreadsheet design in some way. Typically, these applications still exhibit three properties, in common with spreadsheet programs (from [Hud94]):

- visibility of all intermediate results;

- continuous execution;
- integration of input, output, and “program”.

‘NoPumpG’ by Lewis [Lew90], which was followed by ‘NoPumpII’ by Lewis and Wilde [WL90], extend the spreadsheet notion to include interactive graphics. A PhD student of Lewis, Repenning, created ‘Agentsheets’ [Rep93], an environment containing a number of autonomous, communicating agents organized in a grid much like a spreadsheet. Rausch built “The Agent Repository” [Rau96] which was the inspiration for our own ‘empublic’ archive [WRB]. Agentsheets is now the basis for a commercial company, AgentSheets Inc [age]. NoPumpG was also the inspiration for the ‘Penguims’ system, a “Programmable ENvironment for Graphical User Interface Management and Specification” intended to be used by end-users for user interface customisation [Hud94]. Forms/3 [BAD⁺01] is a prototype interactive environment constructed with the intention of testing the limits of the “spreadsheet paradigm” as defined by Kay’s value rule.

Moving away from spreadsheet programs, dependency is also a concept central to database systems, where a view is linked indivisibly to some table data. Database tables behave as literal values do in a spreadsheet — their value does not change unless they are assigned to. Database views behave as spreadsheet formula do — their value changes along with their source values. Typically, the data contained in a database view is evaluated only on demand — for example, only when the `SELECT` command is given in the sequence of SQL commands below.

```
CREATE VIEW name_only
  AS SELECT fname, lname
     FROM customer;
SELECT * FROM name_only;
```

In the most basic configuration, a *materialized view* (which is created using the `CREATE MATERIALIZED VIEW` syntax in Oracle SQL) behaves as a database table: it is a ‘snapshot’ of data as evaluated at the time the materialized view was created and is not updated when source values change. However, a materialized view can be configured to act as a standard view (using `REFRESH ON COMMIT` in Oracle SQL), being automatically updated whenever source values change, or other possibilities in the continuum between view and table — for example, being automatically updated

```
pgm: codegen.o parser.o library
    cc codegen.o parser.o library -o pgm    # load

codegen.o: codegen.c definitions
    cc -c codegen.c                        # compile

parser.o: parser.c definitions
    cc -c parser.c                        # compile

parser.c: parser.y
    yacc parser.y                          # generate parser into
                                           # file y.tab.c
    mv y.tab.c parser.c                    # change name of output
```

Listing 1.1: A description file for the `make` tool, taken from [Fel79]

once per day. A materialized view is typically used to pre-calculate data required for expensive queries, perhaps during an off-peak time of database use. As it uses pre-calculated data, a query using a materialized view is likely to give a faster response time. Gupta *et al* [GMR95] consider how to efficiently update materialized views when source data changes. Halevy [Hal01] survey the techniques available for automatically rewriting queries to transparently use hidden materialized views in order to improve response times for standard queries.

We now draw attention to use of the dependency concept for software development. Firstly, many tools exist to direct code compilation to maximise efficiency and minimise response time, effectively propagating source file modifications through to compiled executable files by invoking a sequence of tool chain actions, guided by file system timestamps and a graph of dependency information. The UNIX tool ‘`make`’ is an early and well-known example of this technique. Listing 1.1, taken from the original paper on the `make` tool by Feldman [Fel79], shows a description of the dependency between targets (for example, the output executable file `pgm`) and sources (for example, the input file `codegen.o`, which also happens to be a target of the second rule). The dependency information is augmented with a description of a shell script action, which, when invoked, will make the target file consistent with the source file.

Many variants of the `make` tool are now common, including versions that can

perform actions in parallel (e.g. [Baa88], [FH89]) or even optimistically, ahead of time [BZ89]. The original `make` tool is independent of source and target file language, but uses shell script syntax extensively for description of actions. Similar tools also exist that are targeted towards a particular source language (for example ‘Jam’ is designed for C and C++ [WS97, Per]), or that use a different language to describe actions (for example, Apache ‘Ant’ [ant] is extended using Java classes to configure actions). These facilities are also commonly incorporated into Integrated Development Environment (IDE) tools, and even compilers [CP03].

More abstract than a tool, but still with influence over software development, the Model-View-Controller pattern used for graphical user interfaces ([KP88], [BMR⁺96, pp.125–143]) separates processing, output and input respectively, joining them again using dependency. Typically the Publisher-Subscriber pattern [BMR⁺96, pp.339–343] is used for change propagation here.

A more concrete application of dependency to graphical user interfaces is demonstrated by Vander Zanden and Myers *et al*’s ‘Garnet’ and ‘Amulet’ graphical toolkits [VHM⁺01] that use one-way dataflow constraints. (Heron [Her02], an EM-related MSc thesis, contains many references to a “dependency tracker” by Perry [Per01] that performs a similar task.) The systems appear similar to the Penguins system mentioned earlier, but the toolkits are intended for use by developers, rather than end-users, in order to create interactive graphical applications. The Garnet and Amulet systems also incorporate pointer variables, which, when referenced in a constraint (formula), form an indirect reference constraint. This feature brings the problems of indirect forms of reference described earlier in respect of spreadsheet program implementation. Vander Zanden *et al* [VMGS94] describe the uses for this feature and various implementation strategies.

The above references generally assume a sequential uniprocessor environment. Bharat [BH95] presents a one-way constraint satisfaction algorithm that is distributed and concurrent. It is based on the sequential algorithm used in the Penguins system. It aims to meet the following guarantees:

- Each participant sees changes in the order they occur.
- Each participant always sees the cause of a modification before the effect.
- Whenever a colleague communicates with you, your view shows every change that theirs does.

Another way to consider the problem of efficiently updating spreadsheet-like state after a change is as an “incremental computation” problem. The language ‘INC’ [YS91] was designed to make it easy to write “incremental programs”, which are performed repeatedly on nearly identical inputs. Ramalingam and Reps [RR93] give a guide to some of the literature in this area. The ‘incXSLT’ incremental transformation framework [VL02], a proposal for XML document manipulation, is a more recent example paper on this theme. Not all agree that the term ‘incremental’ is an appropriate label in this context: the recent paper by Acar, Blelloch and Harper [ABH02] on “Adaptive Functional Programming” uses the word ‘adaptive’ as an alternative for ‘incremental’.

In the software literature, it now seems agreed that the idea of “dataflow languages” is subsumed within “functional languages” [Hud89, p.380]. However, dataflow *hardware* has a long and rich heritage, which was recently reviewed by Šilc *et al* [ŠilcRU01]. They conclude by critiquing the curious present situation, where the von Neumann interface to the processor forces much translation between dataflow and sequential code:

Why should:

- a programmer design a partially ordered algorithm, and
- code the algorithm in total ordering because of the use of a sequential von Neumann language,
- the compiler regenerate the partial order in a dependence graph, and
- generate a reordered “optimized” sequential machine code,
- the microprocessor dynamically regenerate the partial order in its out-of-order section, execute due to a micro dataflow principle,
- and then reestablish the unnatural serial program order in the completion stage?

They suggest that the trend in dataflow research has been to incorporate explicit notions of state into architecture, and that, at the same time, von Neumann computing has incorporated many of the dataflow techniques, the multithreaded model of computing emerging within the resulting continuum.

The main ideas of spreadsheet programs and the `make` tool are, of course, well-known. The purpose of this section has been to document their development in more detail than has appeared before, either in EM-related theses or elsewhere. Certain other ideas and techniques, for example those related to the topic of materialized

views and those discussed in [VMGS94], appear to be related to some of the technical work described in this thesis. They have come to the author's attention too late to have the detailed analysis they deserve, and will be a focus of future research. The distinctive feature of the technical work described in this thesis is that it is conceived within the broader framework of "Interaction with Meaningful State". The concept of meaning here embraces personal understanding that arguably cannot be expressed in a formal and objective fashion but is implicit in the interaction between the modeller and the model. This can be best appreciated by contemplating the richness of the agency invoked in modelling with the Authentic ADM (*cf.* Chapter 2) or the `dtkeden` interpreter (*cf.* §4.1.6).

1.6 Thesis aims

This introduction has given a brief overview of Radical Empiricism and possible uses of Empirical Modelling. The thesis itself is primarily about the questions posed on p.2:

- How can we best exploit and develop our existing tools for implementing EM activity?
- What prospects are there for better tools in the future?

These questions involve significant technical issues but are also foundational and therefore somewhat philosophical in nature.

The EDEN tool used for the planimeter model is now a 30,000 line C program. It implements dependency but is itself constructed from a non-trivial amount of procedural agency. Considering that the von Neumann machine it executes on is constructed partially from combinatorial logic circuits, it is a valid question to ask if dependency can be implemented and used at a lower level.

A long term aim (which goes beyond this thesis) is therefore: to construct a basic microcomputer which implements and uses dependency at a low level. Interaction with the machine should allow questions such as "why is that pixel white?" to be answered using techniques similar to those that might be applied to a mechanical

typewriter. In such a machine, internal state would be linked with dependency, giving it meaning beyond the raw von Neumann state.

This thesis works towards this long term aim by attempting to answer relevant smaller questions: How can we improve our EDEN tool? Can we replace some of the 30,000 lines of C with a smaller core that implements dependency? What is this small core? What precisely is dependency? And a definition?

On current digital computers, in order to implement dependency, it is first necessary to specify the agency that is involved in definition maintenance. This can be seen as adopting a ‘dependency-as-agency’ perspective. This motivates the final aims of the thesis, which are: to elucidate the difference between dependency and agency with respect to a digital computer, and to determine a framework enabling the two worlds of dependency and agency to be bridged. If this is done, we have taken the first steps towards enabling interaction with meaningful state on a digital computer.

This introduction has divided the topic of dependency into four sections organised ‘top-down’. However, both the division and the top-down organisation were primarily for the purpose of exposition. This thesis takes a bottom-up view of the topic that is unprecedented in previous EM literature.

The concept of dependency, together with the intimately related concepts of observable and agency appear to have wide-ranging applicability. Examples of dependency have included:

- the possibility of a “missed free-throw” in a basketball tournament is linked to the size of the basketball;
- in the Phillips machine, the surplus balance in the economy is represented by the level of water in a particular tank;
- the reservation status of a seat on a particular scheduled future aeroplane flight is recorded in a database;
- a document icon in a GUI is used as a metaphor for the document’s contents;
- the wheel rotation of a planimeter is linked to the rotation of the disk and the cantilever position;

- the colour of a time slot in our computer ‘timetabling instrument’ ISM can be linked to the availability of the slot;
- the external state of a digital watch ISM is linked to the internal state as described by a state chart;
- the national income is linked in a spreadsheet to the consumption parameter and marginal propensity to save;
- the output of a digital logic gate or circuit (feedback not considered) depends upon the current inputs.

These examples illustrate the broad range of perspectives that our dependency concept is intended to envisage: the examples here relate to all aspects of explicit, situational, mental and internal (SEMI) state [BRWW01]. In this thesis, we will be concerned with the implementation of dependency with reference to all these perspectives.

1.7 Thesis outline

This thesis contains six chapters. The outline is as follows.

Chapter 1 is this chapter, which motivates and states the aims for the thesis as well as providing an overview.

Chapter 2, “States and transitions in the ADM”, describes the concepts of the LSD account and a machine developed to animate such accounts, the Abstract Definitive Machine (ADM) [Sla90]. Slade describes three possible strategies that can be used in the implementation of a definition maintainer. The ADM is an example of the first strategy: “evaluate at every use (storing only formulae)”. The main emphasis of the ADM is *agency*. The critical review reveals subtleties and inconsistencies in the way that the ADM concept has developed over time. An alternative algorithm for ADM execution is formulated and termed the ‘Authentic’ ADM. The notion that states and transitions can be placed into ‘major’ and ‘minor’ categories, depending upon the perceived granularity, is also developed, which is used later in the thesis. The appendix to Chapter 2 demonstrates the use of Slade’s ‘am’ implementation of the ADM.

Chapter 3 examines the Definitive Assembly Maintainer (DAM) machine [Car99]. The DAM machine is an example of the second evaluation/storage strategy: “evaluate at each redefinition (storing formulae and values)”. A topological sort algorithm [Knu73] is used to schedule calls to operators. The system is written in ARM assembler. A graphical front-end ‘!Donald’ uses the DAM machine to perform definition maintenance in an implementation of the “Definitive Notation for Line Drawing”, DoNaLD. The DAM machine emphasises only *dependency*, and so graphics must be created by operator side-effect. The chapter shows how the DAM machine and !Donald were extended to allow low-level definitive programming in ‘DAMscript’ and also how the system was extended to show the state of the DAM machine store directly on screen, when the video hardware can be considered to be maintaining dependency between the DAM machine store and screen pixels. Three character glyphs are created on the screen using dependency. In this example, “why is that pixel white?” is a question which can be answered through investigation of a script graph of indivisible, ‘instantaneous’ and statically traceable relationships, lending the answer more meaning than would be the case without a definition maintainer. The appendix to Chapter 3 explains my concept of the ‘script graph’: this is an acyclic directed graph representing the dependency in a definitive script. An enumeration of the possible script graphs for $N=4$ and $N=5$ is presented.

Chapter 4 examines the Evaluator of DEfinitive Notations (EDEN) [Yun90]. EDEN has been the primary tool of the research group at Warwick since 1987. It is explained after the ADM and the DAM machine due to the greater complexity of the tool, which gives equal emphasis to both dependency *and* agency. An overview of the historical evolution of the tool is given. The first Eden model (*texteditorYung1987*) is used to illustrate EDEN’s dependency and agency features. An overview of developments since 1999 involving the author is given, which includes work on the theme “constructing EDEN in itself”. The chapter then contains a close study of EDEN’s scheduling and execution mechanisms, firstly from perspective of a tool user, who must treat the tool as a black box, interacting by using notations only. The virtual machine used by EDEN is explained and the scheduling algorithm is summarised in pseudo-code (given in the appendix to Chapter 4) and two diagrams. These results are used to explain the differences between dependency and agency in EDEN.

Finally Y.P. Yung's proposal [Yun96] for a re-implementation based on a Four Layer Prioritised Action Scheme is briefly examined.

Chapter 5 investigates three deep problems in dependency maintenance that run through the earlier chapters in the thesis. The Eden language allows the distinctions between dependency and agency to be illustrated, but the EDEN implementation is firmly rooted on a sequential machine. The notion of the script graph is extended by adding agents to implement dependency-as-agency. There is more than one possible way to add agents to a script graph, and this is outlined. The resulting definition-agents can in principle operate concurrently. The question of how they should be synchronised is examined and a prototype solution in the 'SR' concurrent language is given in the appendix to Chapter 5. Some problems with definitive lists are revealed and the related theory of moding [Mez87] examined. The `%edens1` notation is developed as a prototype solution to problems with definitive lists. A 'tri-box' framework for concurrent dependency maintenance is presented, based on coordination between Observing, Changing and Updating agents. The script graph is represented by 'boxes' in store which can themselves be maintained by dependency, giving a sound basis for Higher Order Dependency.

Chapter 6 gives some conclusions, outlines the contribution of the thesis and contains some brief statements about possible future work.

Chapter 2

The Abstract Definitive Machine, ADM

The ADM is the Abstract Definitive Machine. The name itself makes a substantial claim. If we are to investigate the fundamentals of dependency and indivisibility, the ADM would seem a good place to start.

It is 16 years since the first writing about the Abstract Definitive Machine (ADM) was published. This chapter reviews the ADM in the light of subsequent research (see §2.5 for more details of the background sources consulted). A significant contribution of this review work has been to reveal subtleties and inconsistencies in the way in which the ADM concept has been presented and developed since it was first fully described by Slade in [Sla90]. Particularly significant is the fact that, since the early publications over the period 1988–1990 by Slade, Beynon *et al* that focussed on the ADM as an abstract machine model for parallel “definitive programming”, the concept of the ADM as a *machine* has essentially been neglected. As evidence of this, Rungrattanaubol devotes a chapter of her PhD thesis [Run02, §3] to the “Abstract Definitive *Modelling* framework”. During my critical review, it became clear that there is a significant difference between the framework described by Rungrattanaubol and the original ADM concept as described by Slade. The difference is significant enough to require (in my opinion) a totally new name for the framework as described by Rungrattanaubol (see §2.3.4).

In this chapter, we first attempt to clarify the concepts of the LSD account

and the ADM (in its original concept as described by Slade) and the distinctions between them. We then briefly review the existing implementations and describe implementation strategies that determine the organisation of Chapters 2, 3 and 4 of this thesis. In §2.3 we investigate the operational semantics of LSD and the ADM. A few small clarifications to the existing literature are found to be necessary. Section §2.3.4 then briefly reconsiders Slade’s ADM as clarified and our requirements for an EM tool today, and identifies a discrepancy between the ADM as characterised in [Sla90] and Beynon and Slade’s original intentions for use of the ADM in conjunction with LSD. This leads me to formulate an alternative algorithm for ADM execution for which the term ‘Authentic’ ADM is adopted. The context is then clear for a short investigation of the relationship between the ADM and Chandy and Misra’s UNITY in §2.4. Finally an overview of the relevant sources is given in §2.5.

2.1 Concepts of the LSD notation and the ADM

2.1.1 LSD accounts

The ADM was primarily developed as a tool for the animation of LSD accounts. Most sources describe the ADM in the context of LSD. We must therefore briefly review the concept of the LSD account.

The LSD¹ notation (a Language for Specification and Description) [Bey87a] was originally conceived as a semantic model for the CCITT standard “Specification Description Language” (SDL). It was developed in 1986 by Meurig Beynon in collaboration with Mark Norris, then at the British Telecom Research Laboratories. LSD was intended to provide a medium for describing systems at high levels of abstraction, integrating functional and procedural models and synchronisation mechanisms. The first case studies were in fact of telephone systems, but the notation has since been applied in more situations and the underlying philosophy developed.

An LSD account does not have a formal operational semantics (hence the usual accompanying use of the word “account” rather than “specification”). It serves to describe aspects of interactions amongst agents that are identified in the preliminary understanding of a phenomenon or a domain. This corresponds to the usual starting

¹In part so-called since LSD meant pounds, shillings and pence for its originator.

point of a system builder with an imprecise requirement, where the objective status of much of the domain is not yet known. The LSD account is intended to offer support at this initial, pre-operational, stage of understanding.

An agent description in LSD takes the syntactic form shown in Figure 2.1. The figure is the first attempt to be made in print of an in-depth description of the essential properties of LSD syntax. This is a difficult task as the syntactic conventions changed several times over the period 1986–1992 and previous sources have not made the distinction between the essential ingredients and any extensions² necessary in a particular domain and modelling context. I have constructed the figure from the more recent of the sources mentioned at the end of this section, and have taken the liberty of following Y.P. Yung’s suggestion [Yun93, p.144] of renaming the term ‘protocol’ to ‘privilege’. Yung does not himself follow his proposal, in order “to prevent too many versions of LSD notations being in circulation”, but we feel now (10 years later) that clarity rather than conformity is the aim.

Using LSD, an agent may be described from the point of view of the modeller in the following terms (developed from [BCSW99]):

1. An identity, established by *agent_name* and the values of *parameter_list*;
2. **state** observables: Observables owned by the agent (in the sense that when the agent is absent, its state observables do not exist);
3. **oracle** observables: Observables that are deemed to influence the behaviour of the agent;
4. **handle** observables: Observables that the agent can conditionally (re)define during the course of an action;
5. **derivate** observables: Declared dependencies between observables³ that are projected to hold in the view of the agent;
6. **privileges** for action: Guarded actions that describe circumstances under which state-changing actions *can* (but not necessarily *will*) be performed.

²Examples include the use of types in Bridge’s LSD specification of a vehicle cruise controller in [BBY92] and agent roles in Beynon *et al*’s LSD account of a digital watch in [BCSW99].

³These observables need not necessarily be **oracles** of the agent.

```

agent      ::= agent agent_name ( parameter_list_description ) {
                [ state obs_name_list ]
                [ oracle obs_name_list ]
                [ handle obs_name_list ]
                [ derivate derivate_list ]
                [ privilege action_list ]
            }

obs_name_list ::= obs_name ( , obs_name ) *
derivate_list ::= derivate ( , derivate ) *
action_list  ::= action ( , action ) *
obs_name     ::= string
agent_name   ::= string

action       ::= guard → command_list
guard        ::= boolean_formula
command_list ::= command ( ; command ) *
command      ::= redefinition | instantiation | deletion
instantiation ::= agent_name ( [ parameter_list ] )
deletion     ::= delete agent_name ( [ parameter_list ] )

derivate     ::= obs_name = formula
redefinition ::= derivate |
                obs_name = formula_with_evaluation

```

Notes:

1. The syntax of *formula*, *boolean_formula* and *formula_with_evaluation* are undefined, in order to allow for many possible type algebras. An exception to this is that the vertical bar ‘|’ syntax is used to denote the evaluation of a sub-expression in *formula_with_evaluation* only.
2. The syntax of *parameter_list* and *parameter_list_description* are defined, but are omitted from this particular description.
3. An observable named `LIVEagent_name` represents the live-ness of the named agent.
4. The *obs_names* exist in one “global name space” (to borrow a programming term). However, an observable is not observable/changeable by an agent unless marked as an `oracle/handle`.
5. In many contexts it is reasonable to assume for example that `state` implies `oracle` and `handle`, and that `derivate` implies `oracle` but no such implications are valid in general and so the status of observables should be explicitly stated in an account.

Figure 2.1: BNF describing syntax for an LSD account

Beynon *et al* [BCSW99] go on to describe the intended breadth of description, expanding on the note above about privileges representing *can* but not *will*:

The LSD account is not to be mistaken for a formal specification of system behaviour. It is concerned only with how state-changing actions are attributed to agents, and how their interaction is mediated through observables at their interfaces. The context for agent interaction, and the viewpoint of an objective external observer are conspicuously absent.

The development of an LSD account of a phenomenon is intended to proceed in parallel with the construction of an EM model. Beynon [Bey97] describes the way in which a modeller’s view of an agent may develop in the course of modelling, with reference to “three concepts of an agent within a unifying framework for model construction”:

- View 1:** an entity comprising a group of observables with unexplored potential to affect system behaviour;
- View 2:** a View 1 agent capable of particular patterns of stimulus-response within the system;
- View 3:** a View 2 agent whose pattern of stimulus-response interaction can be entirely circumscribed and predicted.

Composing an LSD account allows us to more precisely describe our current understanding in terms of observables, dependencies and agency. Firstly we make a provisional decision about the number and identities of agents present — as represented by clusters of state observables. We then attempt to add detail to the account by classifying observables as oracles, handles and derivatives and associating privileges to agents. The process of composing the account may itself suggest revisions, or experiments to be performed with the referent or EM model, which then provide further experience requiring revision of the account. Our understanding of agents may thus develop beyond view 1 perhaps as far as view 3, if the domain lends itself to such characterisation and our understanding is developed enough.

An excerpt from an example LSD account is shown in Figure 2.2, which is taken from [Yun93, Appendix H.1]. The figure shown is an excerpt from Y.P. Yung’s account of a railway, including train, driver, guard, station-master, passengers and carriage doors. We will return briefly to this LSD example in §2.3.1.

This brief introduction to LSD will suffice here: for further detail, the development of LSD can be traced through [Bey87b] (= [Bey87a]), [BN87], [BNS88],

```

agent passenger((int) p, (int) d, (int) _from, (int) _to) {
// passenger p is intending to travel from station _from to station _to
// and he will access through door d of the train
state    (int) from[p] = _from;
         (int) to[p] = _to;
         (int) pat[p] = _from;
         (int) door[p] = d;
         (int) pos[p] = 2;
         (bool) alighting[p], boarding[p], join_queue[p,d];
oracle   (int) at, pat[p];
         (bool) queueing[d], pos[p], door_open[d];
handle   (int) pos[p], pat[p];
         (bool) door_open[d];
derivate alighting[p] = at == pat[p] ^ at == to[p] ^ -2 ≤ pos[p] ≤ 0 ^ engaged;
         boarding[p] = at == pat[p] ^ at == from[p] ^ 0 ≤ pos[p] ≤ 2 ^ engaged;
         join_queue[p,d] = (alighting[p] ^ door_open[d] ^ pos[p] == -1) ||
                           (boarding[p] ^ door_open[d] ^ pos[p] == 1);
         LIVE = ¬(pat[p] == to[p] ^ pos[p] == 2);
privilege boarding[p] ^ pos[p] == 2 → pos[p] = 1;
         alighting[p] ^ pos[p] == -2 → pos[p] = -1;
         alighting[p] ^ ¬door_open[d] → door_open[d] = true;
         alighting[p] ^ pos[p] == 0 ^ door_open[d] ^ queueing[d]
           → pos[p] = 1; pat[p] = lat; pos[p] = 2;
         alighting[p] ^ pos[p] == 0 ^ door_open[d] ^ ¬queueing[d]
           → pos[p] = 1; pat[p] = lat; door_open[d] = false; pos[p] = 2;
         boarding[p] ^ ¬door_open[d] → door_open[d] = true;
         boarding[p] ^ pos[p] == 0 ^ door_open[d] ^ queueing[d]
           → pos[p] = -1; pat[p] = at; pos[p] = -2;
         boarding[p] ^ pos[p] == 0 ^ door_open[d] ^ ¬queueing[d]
           → pos[p] = -1; pat[p] = at; door_open[d] = false; pos[p] = -2;
}

agent door((int) d) {
state    (bool) queueing[d], occupied[d], around[d];
         (bool) door_open[d] = false;
oracle   (int) pos[p], door[p]; (p = 1 .. number_of_passengers)
         (bool) join_queue[p,d]; (p = 1 .. number_of_passengers)
handle   (int) pos[p]; (p = 1 .. number_of_passengers)
derivate queueing[d] = there exists p such that join_queue[p,d] == true;
         occupied[d] = there exists p such that (pos[p] == 0 ^ door[p] == d)
         around[d] = there exists p such that (door[p] == d ^ -1 ≤ pos[p] ≤ 1)
privilege queueing[d] ^ ¬occupied[d] ^ join_queue[p,d] → pos[p] = 0; (p = 1 .. number_of_passengers)
}

```

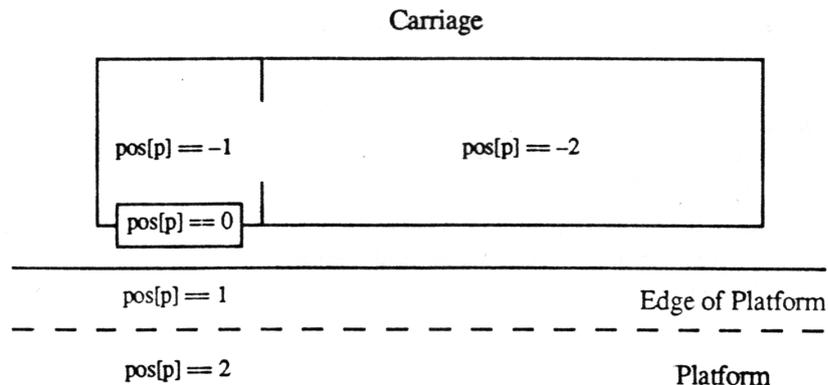


Figure 2.2: LSD account of a train carriage door and passenger (from [Yun93, Appendix H.1])

[BNOS90], [Sla90, §5.1], [BBY92] (which briefly describes the ‘modern’ version) and [Yun93, p.142].

2.1.2 1-agent modelling with the ADM

In [Sla90], Mike Slade presents the Abstract Definitive Machine (ADM), “a computational model for definitive programs”. The ADM enables interaction with meaningful state through the provision of a *definitive notation*, which allows the formulation of a definitive script⁴ to describe state and the introduction of redefinitions to make transitions manually.

The ADM uses *definitions* to describe state. A definition can be described with the following simple BNF:

$$\textit{definition} ::= \textit{var_name} = \textit{formula}$$

It describes the association of the *formula* with the variable named by *var_name*. The formula establishes a functional relationship between variable values. This relationship is expressed using operators from a chosen underlying algebra. The formula represents a recipe for determining the value of the variable, and means that changes in the values of variables in the formula will in general affect the value of the variable. Following Slade [Sla90, §2.1.1], a state described by a system of definitions is called a *definitive state*.

A *transition* to a new definitive state is caused by redefining a variable, which changes the formula associated with it. Redefinitions can take the same form as definitions, with an optional extension including evaluation.

$$\begin{aligned} \textit{redefinition} ::= & \textit{definition} \mid \\ & \textit{var_name} = \textit{formula_with_evaluation} \end{aligned}$$

The same notation can therefore be used to describe the state and transitions upon the state. The current state is described by a set of definitions. A transition to another state is described by a set of redefinitions.

⁴Slade describes input to the ADM as a “definitive program” — we now prefer the term “definitive script” (a term not yet coined at the time of Slade’s writing), which is intended to emphasise *state* over behaviour.

The current state as described by the set of definitions is ‘timeless’. Although it is arrived at through a history of interaction, only the current formulae are needed to describe the current state. A redefinition however occurs at a particular point in time. Making a redefinition is an act of agency — it is a *command*.

As a redefinition occurs at a particular point “in time”⁵, a formula within a redefinition may meaningfully contain evaluation (which is denoted by surrounding an expression with ‘| ’ vertical bars in the ADM script syntax). Slade [Sla90, §2.1.1] gives the following example of a redefinition:

```
new_currency = |exchange_rate| * amount
```

resulting in the definition formula associated with `new_currency` becoming set to:

```
new_currency = 5.22 * amount
```

if the value of `exchange_rate` at the time of redefinition is 5.22.

The ADM provides a way of partitioning state that is related to the description of an agent in an LSD account. Definitive variables are organised into named groups called *entities*. A *program store* P holds entity descriptions. Entities are instantiated from their description in P , whereupon their definitive variables are added to the definition store D , which holds definitive variables that are *owned* by currently instantiated entities. Several further *commands* are provided by the ADM in order to manage the descriptions of entities in P and to instantiate them in D . An example will be given at the end of the next section, which explains how the ADM may be ‘programmed’.

The instantiation of entities in D and A from descriptions in P is analogous to instantiation of objects from classes in object-oriented programs (of which the first example was SIMULA 67 [BDMN79]), but the ADM does not provide the now-common full object model, with a class hierarchy and inheritance.

In principle, use of dependency in the ADM extends to output. Slade describes how this may be envisaged [Sla90, §2.3.2]:

A variable called `output` is used to provide output facilities. The variable is linked to the output device in such a way that changes to the variable cause changes in the state of the output device. The changes that are made to `output` should be consistent with the nature of the output device.

⁵More precisely, in a particular definitive state.

If the output device consists of eight LEDs, then an appropriate change to the value of `output` would be to define it by a formula which always evaluated to an integer between 0 and 255. The current value of `output` would at all times be interpreted as an eight digit binary number which indicated which LEDs are to be illuminated. This use of `output` involves defining it as a function of the current state, so that as the state changes so will its value and the state of the output device. The definition of `output` will be of the form

$$\text{output} = \text{function of state}$$

I demonstrate a very direct form of this kind of ‘definitive output’ in §3.5.4, where the video hardware of the DAM machine is used to directly display internal definitive state.

2.1.3 ‘Programming’ the ADM

The previous section described how the ADM provides definitive state and allows the modeller to make transitions using redefinitions. Using the facilities described so far, no change occurs to the definitive state unless the modeller makes a redefinition — hence, the ADM supports 1-agent modelling.

The ADM also implements automated state-change, or *automated agency*. In the ADM, each state-changing *action* is conceptually made autonomously by an entity when a particular enabling condition is detected in the state. The enabling condition for an action is described by a *guard*. ADM scripts are therefore similar to LSD accounts, in which a privilege for a certain action is expressed by the use of a guard. However, a guard in LSD describes only a necessary condition: when the guard is true, the LSD action *can* be performed. A guard in the ADM is closer to the spirit of [Dij76]: when the guard is true, the action *will* be performed.

In the ADM, an action is described using the following BNF⁶.

$$\begin{aligned} \text{action} &::= \text{guard} \rightarrow \text{command_list} \\ \text{guard} &::= \text{boolean_formula} \\ \text{command_list} &::= \text{command} (; \text{command})^* \\ \text{command} &::= \text{redefinition} \mid \text{instantiation} \mid \text{deletion} \end{aligned}$$

⁶Note the use of the symbol \rightarrow which does not appear on a conventional keyboard — the syntax given here is for the ADM, not an implementation.

```

entity one(first) {
  definition
    var1 = first,
    change1 = (var1 > var2)
  action
    change1 → var1 = |var1 - var2|,
    !change1 && !change2 → output = |var1|; delete one(first)
}

entity two(second) {
  definition
    var2 = second,
    change2 = (var2 > var1)
  action
    change2 → var2 = |var2 - var1|,
    !change1 && !change2 → delete two(second)
}

```

Listing 2.1: Two ADM entities that can compute GCD

Slade [Sla90, §2.1.3] gives the following examples of actions:

```

¬ can_start → choke = true;
(time == 2000) → switch = false; alarm = true;

```

Actions are mapped to entities in a similar manner to definitions. The entity descriptions in the program store P contain information about the potential actions performed by each entity. When the entity is instantiated from P , its actions are added to the *action store* A , which holds action descriptions that are owned by currently instantiated entities.

Listing 2.1 (taken from [Sla90, §3.7], with minor changes from `am` to ADM syntax) gives an example of an ADM script containing two entities that interact to eventually calculate the GCD of two integers (provided as parameters to the entities when they are instantiated) in a form of a “dance”⁷. Figure 2.3 gives the BNF that describes interaction with the ADM. I have constructed the figure by reference to [Sla90] (who gives a detailed description of the syntax accepted by his `am` implementation but no complete syntax for ADM scripts). The particular algebra used is not specified

⁷[BR92] gives details of a “folk-dance routine” for computing GCD.

in the ADM, and so the syntax for expressions is undefined, aside from the vertical bar ‘|’ syntax for evaluation. Figure 2.4 (which is based on [Run02, Figure 3-4, p.82]) illustrates the structures contained in the ADM during use.

```

statement ::= command | query | entity

command ::= redefinition | instantiation | deletion

redefinition ::= definition |
                 var_name = formula_with_evaluation
definition ::= var_name = formula

instantiation ::= entity_name ( [ parameter_list ] )
deletion ::= delete entity_name ( [ parameter_list ] )

query ::= ? var_name

entity ::= entity entity_name ( [ parameter_list_description ] ) {
                [ definition definition_list ]
                [ action action_list ]
            }

definition_list ::= definition ( , definition )*
action_list ::= action ( , action )*

action ::= guard → command_list
guard ::= boolean_formula
command_list ::= command ( ; command )*

var_name ::= string
entity_name ::= string

```

Notes:

1. The syntax of *formula*, *boolean_formula* and *formula_with_evaluation* are undefined, in order to allow for many possible type algebras. An exception to this is that the vertical bar ‘|’ syntax is used to denote the evaluation of a sub-expression in *formula_with_evaluation* only.
2. The syntax of *parameter_list* and *parameter_list_description* are defined, but are omitted from this particular description. Further description of the issues surrounding parameter lists is given in [Sla90, §2.3.1].

Figure 2.3: BNF describing syntax for the ADM

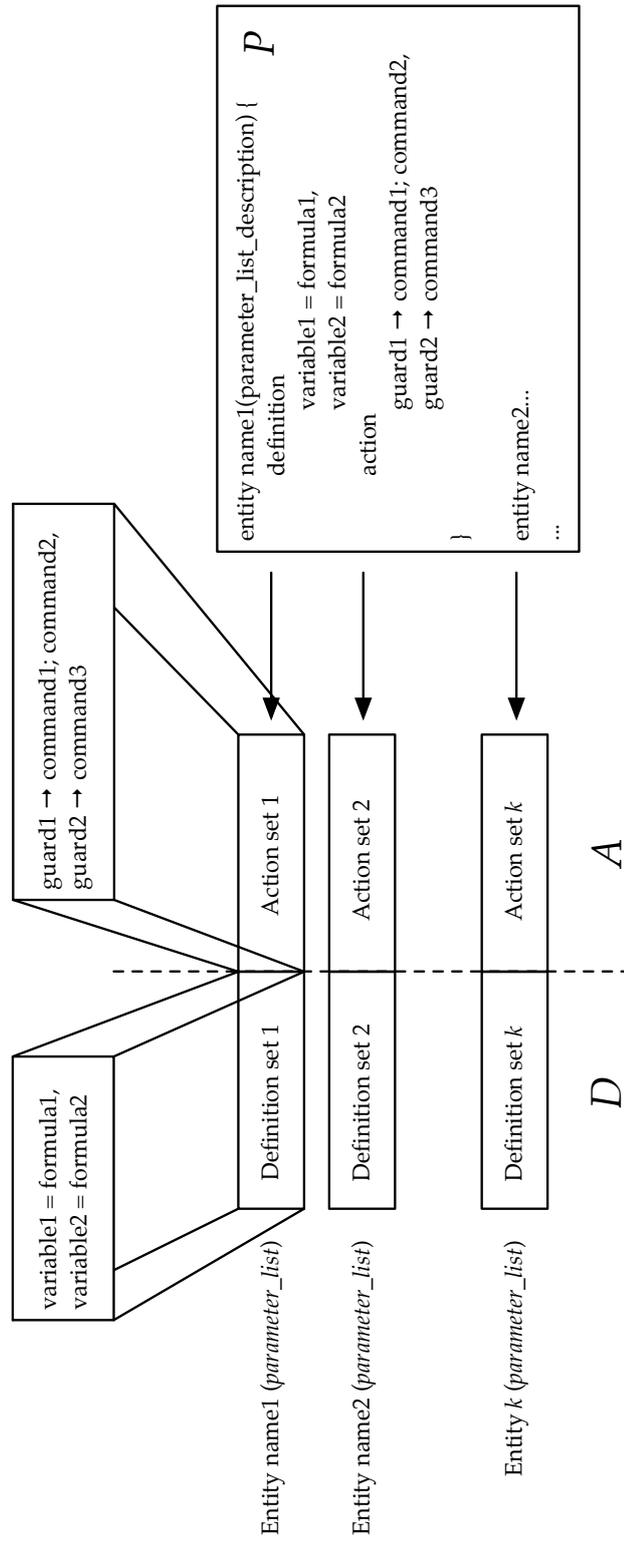


Figure 2.4: *D*, *A* and *P* in the ADM

2.1.4 From LSD accounts to ADM scripts

There are three significant differences between LSD accounts and ADM scripts. They reflect the fact that LSD was designed to model experience from multiple (possibly inconsistent) points of view, and the ADM was designed to model experience from the perspective of an external observer.

Perception or direct experience of values: There is no global state in LSD.

In this respect, it is a “distributed computing model”. An LSD agent with a `state` observable carries the *authentic* value of the observable. Agents can access the values of `oracle` observables only by *perceiving* them.

The independence of different perceptions is emphasised in Slade [Sla90, §5.3.1], who states “The authentic value and any perceived values of the same variable will be stored in private memory locations in different agents.”

The synchronisation between authentic and perceived values is not defined in LSD⁸. In the ADM, however, all state is global. In this respect, the ADM is a “shared-memory computing model”. In EM terms, LSD accounts lack the external observer perspective which ADM assumes.

Guards as ‘can’ or ‘will’: In LSD, guards represent privileges for action. “In LSD a guard being true gives permission for the agent to act to change the state.” [Sla90, §5.2]. In the ADM, however, “all guards which are true in an execution cycle have their associated command list executed: there is an obligation on the entity to execute its command list” [Sla90, §5.2].

Speed and synchronisation assumptions: In LSD, agents operate asynchronously. Their relative speeds of execution are undefined. “In general [agents] will operate at very different rates. . . Part of interpreting the specification as describing the behaviour of a system will involve deciding how fast agents operate relative to each other.” [Sla90, §5.3.2]. LSD guards are also subject to undefined uncertainty: “There can be an arbitrary time interval between the selection of a true guard for execution and the sequential execution of its

⁸Except for in the case of derivatives which refer to `state` observables of another agent, in which case the authentic value is used in “close synchronisation” — see [Sla90, §5.3.1 and §8.5].

Basic concept	LSD	ADM
<i>agent</i>	<i>agent</i>	<i>entity</i> (described in <i>P</i>)
<i>observable</i>	<i>state</i> of agent	<i>variable</i> owned by entity (in <i>D</i>)
	<i>oracle</i> of agent	<i>variable</i> <i>evaluated</i> by action owned by entity
	<i>handle</i> of agent	<i>variable</i> <i>redefined</i> by action owned by entity
<i>dependency</i>	<i>derivate</i> of agent: describes relationship between observables in the view of the agent	<i>definition</i> owned by entity (in <i>D</i>): describes relationship between variables in the view of the entity
<i>agency</i>	action of agent is permitted by a true guard <i>privilege</i>	<i>action</i> (in <i>A</i>) of entity is mandated by a true guard

Table 2.1: LSD and ADM terminology compared

command list” [Sla90, §5.2]. The operation of ADM entities, however, are synchronised by a global clock.

Beynon *et al* [BNOS90] and Slade [Sla90, §5.4] describe how an LSD account may be animated using an ADM script. An LSD account describes the interactions between agents in a concurrent system in terms of their privileges to perform actions. It identifies the characteristics of system behaviour that depend upon the interrelated capabilities and perceptions of its participating agents. A precise description of system behaviour requires additional information, as an LSD account can be given many behavioural interpretations, most of which are inappropriate. An LSD account can be translated into several ADM scripts, each associated with a different scenario for agent action. The terminologies used in the two notations differ in order to highlight these differences. The terms used are compared in Table 2.1.

2.2 Implementations of the ADM

2.2.1 Evaluation/storage implementation strategies for definitive systems

In [Sla90, §3.4], Slade notes that in the ADM, a redefinition does not necessarily imply an immediate evaluation and discusses when definitions should be evaluated, outlining some potential methods. My re-interpretation of the alternatives is shown below.

Strategy 1: evaluate at every use (storing only formulae)

Strategy 2: evaluate at every redefinition (storing formulae and values)

Strategy 3: a mix of 1 and 2: evaluate at use when a redefinition has previously out-dated the store (storing formulae, values and out-of-date flags).

I term these “evaluation/storage strategies” because they each demonstrate a progressive trade-off of storage for evaluation. The strategies are related to the ‘data-driven’ and ‘demand-driven’ classifications of computer architecture, described by Treleaven *et al* [TBH82]:

In data-driven (eg, data-flow) computers the availability of operands triggers the execution of the operation to be performed on them, whereas in demand-driven (eg, reduction) computers the requirement for a result triggers the operation that will generate it.

The strategies are labelled in increasing order of implementation difficulty, assuming the implementation is constructed in a procedural language. Strategy 1 is relatively simple to implement in a procedural language as it corresponds to treating use of definitions as calls to functions⁹. Strategies 2 and 3 respectively use more storage¹⁰ to reduce evaluation, adding more complexity.

Evaluation of just one definitive variable in any of these strategies can be a protracted task, as the variable may use many other nested definitions. The efficiency

⁹For example, the definition `a is b+c;` can be represented by the C function:

```
int a() { return b() + c(); }
```

¹⁰The three strategies are an abstraction involving some loss of detail. Although the `am` implementation uses strategy 1, it needs to store some state to implement evaluation in only an *S* state — see later.

of each strategy in terms of evaluation depends upon the use : redefinition ratio that is present in the script. In strategy 1, evaluation is initiated by use. This strategy is most efficient when redefinition is more frequent than use. In strategy 2, evaluation is initiated by redefinition. This strategy is most efficient when use is more frequent than redefinition. In strategy 3, the work is distributed somewhat across both use and redefinition: the out-of-date flags must be marked on redefinition, and evaluation must be performed at use if the value is found to be out-of-date. A sequence of redefinitions with no intermediate use therefore requires no evaluation, but the implementation must maintain the out-of-date flags. But note that when attempting to characterise the efficiency of this strategy, we are led to consider factors other than purely evaluation. More work and more storage is required to maintain the additional information needed for strategies 2 and 3.

The value of every guard expression in A is checked at the beginning of every transition in the ADM. The appropriate evaluation/storage strategy to use for guard expressions is therefore dependent on the ratio of guard evaluations to guard redefinitions.

The choice of evaluation/storage strategy is a fundamental initial decision taken when a definitive system is implemented and has wide-ranging effect. The three strategies therefore form the organising principle of this chapter and the following two chapters of this thesis. The implementation described briefly below is an example of strategy 1. The DAM machine, described in Chapter 3 is an example of strategy 2. Finally, EDEN, described in Chapter 4 is an example of strategy 3.

2.2.2 Existing implementations of the ADM

The ADM design has been concretely implemented more than once. However, at the time of writing, the implementations are few and all classed as prototypes. They fall into two categories:

- `am`
- `adm`, `adm2`, `adm3`

In this thesis, I will use ADM (uppercase) to denote the Abstract Definitive Machine concept outlined in the previous section, and lowercase implementation

names in Courier type to refer to implementations.

The first implementation to be constructed was named `am` and is presented in [Sla90, §3]. It is a “simulator, in the sense that the commands... are performed sequentially, rather than in parallel”. The source code for `am` is written in the C language and the lex and yacc parser generators are used. The source is approximately 5000 lines of code (comments and blank lines included).

The second way that the ADM has been implemented involves translation from a script written in ADM-style syntax into Eden code. Y.P. Yung presents such an EDEN-based implementation named `adm` in [Yun93, §8.5], and later a second implementation giving more control over evaluation, named `adm2` [Yun96, §6.2]. P-H. Sun presents a third version named `adm3` [Sun99, §6.2.1] which attempts to simplify the syntax of the translated Eden output. Due to the translation, the implementation gains the benefit of Eden’s graphical and interactive features, but the state-transition model becomes hard to analyse in isolation from EDEN’s operational model (the subject of §4.3). These implementations therefore play a minor role in this chapter.

The full BNF grammar for the interaction language of the first implementation, `am`, is given in [Sla90, Appendix 4.2]. It is limited to integer values only but is otherwise quite like the syntax of the language ‘C’. It extends the ADM BNF given in Figure 2.3 slightly, adding the ability to make a “procedural action” when a guard is true. The only available procedural action is a `print` command. This is a simple solution to the problem of output in `am` — the full ADM design (which is currently unimplemented) includes a scheme for input and output that is more consistent with a definitive model, described briefly in §2.1.2. The extension to the ADM BNF applicable in `am` is shown below.

$$\begin{aligned} \textit{action} &::= \textit{guard} \textit{procedural_action} \rightarrow \textit{command_list} \\ \textit{procedural_action} &::= \textit{print}(\textit{string}) \end{aligned}$$

Appendix 2.A (p.91) gives an example of an `am` script and an example interaction, showing how the script can be loaded into the machine, entities instantiated, machine state can be queried, the machine can be invoked and redefinitions can be made.

It should be noted that, in concept, the ADM is substantially more ‘embodied’ than the prototype `am` implementation described here. For instance, Beynon maintains that definitive representations of screen state were within the scope of the

original ADM concept — in which case the choice of the epithet ‘Abstract’ for the ADM is somewhat misleading. In contrast, Slade’s description [Sla90, §8.4.1] of an ADM interface using eight graphical windows under the heading “Extensions to the adm” is oriented towards giving the modeller more control over the execution of an abstract computational model.

The `am` interpreter uses evaluation/storage strategy 1. Slade justifies the choice with the following paragraph [Sla90, §3.4]. I have numbered the sentences for reference below:

[1] There is reason to hope that many programs would consist of redefinitions of the same variable without intermediate evaluations being performed. [2] This is often the case in procedural programs when assignment statements are used to maintain a consistent state, causing unnecessary evaluation since the actual value of the variable is not required in that state. [3] We suggest that this redundant evaluation is caused by the nature of procedural assignments, and can be avoided by the use of definitions. [4] The implementation of the `adm` evaluates variables as needed, although some applications which involve a proportionally large amount of evaluation will be more efficiently executed by using a strategy of evaluating at redefinition time.

Slade’s first sentence is an assumption about ADM ‘programs’. Sentence [2] in effect states that a definition maintainer using evaluation/storage strategy 2 (evaluate-at-redefinition) is inefficient if the assumption holds. Sentence [3] points out that a definition maintainer is free to delay evaluation until use. The last sentence states that this is the strategy used in the `am` implementation, which will be efficient if the assumption holds — if the assumption does not hold, then evaluation/storage strategy 2 may be more appropriate.

2.2.3 A hardware implementation?

Slade [Sla90, §3.4] suggests that the link in von Neumann architecture machines connecting store and CPU, that [Bac78] brands the “von Neumann bottleneck”, poses a limitation in implementing “definitive programming”, since in strategy 2,

Each redefinition will then involve memory accesses to get the values of all dependent variables, computation time to evaluate the formula, and then two writes back to store: the redefinition and its evaluation. Writing a formula is likely to be more expensive than the procedural writing of a value, and the von Neumann bottleneck would therefore become further overloaded with the number of store transitions involved with this technique.

Slade then goes on to suggest a special purpose computer architecture more suited to the execution of “definitive programs”. This text is paraphrased from [Sla90, §3.4]:

If variables’ values are to be available immediately to the CPU, there must be some mechanism to ensure that all dependent variables are updated after each redefinition. We suggest an architecture in which the store is an active participant in the computation process, with the task of maintaining consistency within the state.

When a variable is redefined, the redefinition is stored and the names of all dependent variables (kept as tags associated with each variable) are stacked. Whilst the CPU is involved in other computation (e.g. the evaluation of a guard once the values of all the variables in it have been received from store), the variables on the stack have their values updated by the store processor.

When a read operation is invoked, the stack is checked to see if it contains the variable name. If it does, then the associated definition is evaluated and returned. If it does not, then the value associated with the variable is up-to-date, and so this is returned immediately, without the need for any evaluation.

It is anticipated that the overhead of stack checking will be compensated for by the gains in the time taken to perform evaluations.

This machine may avoid the von Neumann bottleneck, since a write operation will only involve one write access, with no intrinsic read operations as found with value assignments.

It is also consistent with how the hardware of a digital processor actually works: the voltage at any point in a circuit is a function of the inputs to the circuit, and so processing at the lowest level of computation (i.e. circuit level) can be viewed as definitive. The area of hardware development for definitive programming might be a rewarding subject for future research.

This thesis considers how dependency may be implemented on digital computers through study of the ADM, the DAM machine (Chapter 3) and EDEN (Chapter 4). The study gives us more understanding of how dependency might be implemented directly in hardware. In particular, we return to the idea of separating update (performed by Slade’s “store processor”) and change (performed by the various agents/entities) in §5.1.2.

2.3 Operational semantics

2.3.1 Operational semantics of LSD

Y.P. Yung [Yun93, p.144] (actually quoting [Sla90, §5.2]) describes how “the protocol [privilege] of an LSD agent should be interpreted”:

1. All the guards are evaluated.
2. If at least one of the guards is true then a guarded action [with a true guard] is chosen arbitrarily, otherwise the guards are re-evaluated.
3. The action [command] list associated with the chosen guard is executed sequentially.
4. The procedure is repeated.

This algorithmic interpretation adds clarity in two particular semantic aspects described below, but it also contains a mistake.

The mistake in the description is the following: step 2 implies that if exactly one guard is true, the associated action *will* be chosen. However, if this LSD concept is intended to imply a privilege (which Y.P. Yung states in the following paragraph), then step 2 should imply that even though a guard is true, the LSD agent is *not* obliged to choose to perform the associated action. On this basis, a more appropriate specification for step 2 would be:

2. Select a true guard and perform the associated guarded action or return to step 1.

With or without the correction, two semantic aspects are made clear by the description. Firstly, the description makes no reference to context, apart from that pertaining to the guards. This is due to the distinction made in the interpretation of LSD between perceived and authentic values. Whether guards refer to authentic or perceived values is a question that requires addressing.

The second aspect that is made clear by the description is the extent to which an LSD agent can perform parallel action. Steps 2 and 3 restrict an agent so that it only performs one action at a time. Considered in programming terms, LSD is ‘single-threaded’. This restriction stems in part from the original design intention of LSD to represent processes [BN87], but it is also centrally concerned with the role that dependency plays in the modeller’s interpretation of state change. Although only one action is performed at a time by an LSD agent, the associated state-change may affect many observables indivisibly. One purpose of the LSD account is to separate the independent “centres of state change” [BRY90] that determine agency. To date, the application of LSD has focussed upon associating each strand of independent agency with a different agent. This is the simplest type of account. It may be

more realistic in some circumstances to recognise that the same agent is capable of parallel execution of more than one independent action.

Y.P. Yung considers the latter point in [Yun93, §8.4.2]. Some sub-sequences of commands in his LSD accounts are underlined. For example, Figure 2.2 contains the example:

$$\begin{aligned} & \text{privilege alighting}[p] \wedge \text{pos}[p] == 0 \wedge \text{door_open}[d] \wedge \neg \text{queuing}[d] \\ & \rightarrow \underline{\text{pos}[p] = 1; \text{pat}[p] = |at|}; \text{door_open}[d] = \text{false}; \text{pos}[p] = 2; \end{aligned}$$

In English, this example reads: if the passenger is alighting the carriage (getting off the train) and is standing in the doorway and the door is open and there is no queue of passengers in the doorway, then move onto the edge of the platform, take note of the identity of the station, shut the door and move onto the platform proper.

Y.P. Yung [Yun93, §8.4.1] notes of his underlined commands that they “should, in principle, be executed in parallel. . . [but] the current LSD notation has no provision for specifying synchronised actions [commands].” He suggests extending LSD with a new “parallel action [command] separator”¹¹. The underlined commands could then be written:

$$(\text{pos}[p] = 1 // \text{pat}[p] = |at|);$$

Derivates are used in LSD to specify indivisibility in observation from the point of view of an agent. With this extension, the parallel command separator then specifies indivisibility in change made by an agent. Cartwright’s Block Redefinition Algorithm in the DAM machine (see §3.1.2) and Y.W. Yung’s `autocalc` mechanism in EDEN (see §4.3) have the same aim.

Another application for the parallel command separator is illustrated by considering the agency involved in playing a piano. If a scale is being performed, one note is played at a time. This might be represented using the following LSD fragment.

$$\text{staccato_scale} \rightarrow \text{c}=\text{down}; \text{c}=\text{up}; \text{d}=\text{down}; \text{d}=\text{up}; \text{e}=\text{down}; \text{e}=\text{up};$$

If a chord is played, several piano keys may be depressed “simultaneously” and then released. A scale played *legato* involves simultaneous release of one note and

¹¹And goes on to give an example involving the parallel swapping of two observable values, which is not used here due to its formal ‘programming’ connotations.

depression of the next. Using the parallel command separator, these can be represented by the following LSD fragments:

```
chord → (c=down // e=down // g=down); (c=up // e=up // g=up);  
legato_scale → c=down; (c=up // d=down); (d=up // e=down); e=up;
```

The proposed extension to LSD provides some limited scope for parallelism but does not begin to address the complex issues involved in attributing agency in general. For instance, should we regard the playing of a chord as three synchronised actions performed by independent fingers or an atomic action resulting from a single movement of the arm? A more general extension of LSD would allow the description of these two distinct understandings.

2.3.2 Invalid transitions in the ADM

The notion of the *invalid transition*, described by Slade in [Sla90, §2.1.3] is an important feature of the ADM:

A central concept in this work is that the new state which results from the redefinition of a subset of the variables is in general independent of the order of redefinition. The only times when the order of redefinitions can be significant are when the same variable is redefined twice or when a formula in one of the redefinitions involves evaluation. An example of a redefinition involving evaluation is

```
rate_used = |exchange_rate|
```

which represents the fixing of the rate of exchange for a currency conversion. If the variable being evaluated (in this case `exchange_rate`) is also redefined, then the order of the two redefinitions is significant. This example corresponds to the use of an exchange rate at the same time as it is changing.

We call a transition which involves either the evaluation and redefinition of a variable or the redefinition of the same variable more than once an *invalid transition*, since it is not clear which state such a transition would result in...

A set of redefinitions which do not constitute an *invalid transition* can be performed in parallel.

(On this definition, a valid transition is context-independent. Note that there are also types of context-dependent error that can occur in a transition between definitive states: for example, the transition may introduce a graph cycle, although this may not be obvious from an examination of the set of redefinitions alone.)

Brinch Hansen [Han02b, p.22] defines a vocabulary which includes:

concurrent processes, *processes* that overlap in time; concurrent processes are called **disjoint** if each of them only refers to **private data**; they are called **interacting** if they refer to **common data**.

The set of redefinitions in a *valid*¹² transition therefore exhibit the essential characteristic of a set of disjoint concurrent processes, albeit for only that single valid transition.

Brinch Hansen later goes on to state the importance of this concept ([Han02b, p.30], his emphasis):

Hoare introduced the essential requirement that *a programming language must be secure* in the following sense: A language should enable its compiler and run-time system to detect as many cases as possible in which the language concepts break down and produce meaningless results.

For a parallel programming language the most important security measure is to check that processes access disjoint sets of variables only and do not interfere with each other in time-dependent ways.

The ADM is intended to detect an invalid transition at run-time. An invalid transition may be an error fatal to the current automated agency in an ADM ‘program’. However, [Sla90, §2.3.4] continues:

It would be inappropriate to regard an invalid transition... as automatically indicating a flaw in the program being executed, since it may indeed be a faithful modelling of the real world... an invalid transition can be dealt with by specifying beforehand ways of resolving the interference or by allowing user intervention to indicate either what transition is to be performed or by changing the state to one where the resulting transitions are not invalid.

If the run set is valid, it can be executed in parallel and it is guaranteed that the resulting state is well defined. An invalid transition should therefore not be considered “just another sort of error”: the important point is that a valid transition is interference-free. Valid transitions in the ADM therefore provide a powerful guarantee of valid state: since the machine will not proceed with sets of redefinitions that (when acting upon the current state) do not lead clearly to a single possible resultant state, the state is valid at all times. If an invalid transition halts automated agency in the ADM, the state is still valid and if the invalidity is resolved, automated progress can continue if desired.

¹²=Not invalid.

1. Check for cyclic dependency error in D .
2. For each guard in A :
 - (a) Evaluate the guard.
 - (b) If the guard is true, add the associated command list to the run set.
3. If run set is empty, halt execution.
4. Check run set for the various cases of interference error (an invalid transition).
5. Simulate parallel execution of all the command lists in the run set.
6. Go to 1.

Listing 2.2: Slade's ADM algorithm

2.3.3 What's in a transition?

A *transition* in the ADM is an important notion. Much of the semantics depend upon an exact definition of what constitutes a transition. In this subsection, I investigate the notion of the ADM transition in respect of the aspects listed below. The last aspect is a significant theme that runs through this thesis.

- Guard evaluation, Invalid transition
- Command sequencing
- Evaluation

Guard evaluation, invalid transition

With respect to guard evaluation and the identification of invalid transitions, a transition in the ADM is formed from a set of complete command lists. Listing 2.2 shows the algorithm that forms the basis for Slade's `am` implementation. In respect of guard evaluation, a transition is one execution of steps 1–5. Steps 2(b) and 5 make clear that within a single transition, a set of complete command lists are executed.

I have compiled the listing from Slade's algorithm for the ADM execution cycle¹³

¹³It is unclear whether the algorithm presented in [Sla90, §3.6] describes 'the' ADM operational semantics in general or just a particular implementation: Slade states only that it is "an algorithm for the computation. . . [which] describes the execution involved in a single execution cycle. . .". The algorithm is certainly the basis for Slade's `am` implementation. In the absence of any other formal descriptions, here we take it to be a description of 'the' ADM.

[Sla90, §3.6], simplifying by omitting detailed error handling cases.

Guards are not re-evaluated during step 5. Slade makes this clear in the following text, taken from [Sla90, §2.1.3]:

... The definitive state reached after [previous redefinitions within the current command list] cannot be used for evaluating guards in — command lists are treated as a single transition, no matter how many redefinitions they involve.

...

... An adm program consists of a set of actions, and transitions are effected from state to state by executing the commands in command lists. Execution proceeds by evaluating all the guards, placing the commands with true guards in a run set, and then executing in parallel the command lists in the run set.

The computational model described in Listing 2.2 is thus superficially similar to the bulk-synchronous parallel (BSP) model (proposed by Valiant [Val90] in the same year as Slade's work and later developed by McColl *et al*), in that it repeatedly determines the work to be performed in the next 'super-step' (by evaluating guards) and then performs this work (the command lists) in parallel. However, the BSP model does not use definitive state.

Command sequencing

A family of redefinitions can be considered as a sequence or a set if they constitute a valid transition. From an abstract point of view, the ordering of the redefinitions does not matter as there is one well-defined resultant state.

However, the sequencing of commands (which as well as redefinitions, include instantiation and deletion of entities) can be considered significant. Conceptually, we would wish to model a person entering a modelled room as a simultaneous instantiation of the person entity and initialisation of the entity variables. However, in the ADM, the instantiation and initialisation are represented through the use of a sequence. This is the intended purpose of command lists in the ADM, as Slade describes in [Sla90, §2.2.2]:

If there is more than one command list in the run set then all the command lists are executed in parallel. Individual command lists are executed sequentially. This (for example) allows an entity to be instantiated and variables owned by it to be initialised in one command list, or permits an entity instance to perform some final commands and then delete itself.

Ideally we would wish to construct more expressive definitive notations that allow instantiation and initialisation of blocks of state in one redefinition. However, in the

ADM, it seems that Slade found it necessary to implement sequential command lists, intending them to be used only for this limited purpose.

Evaluation

The final aspect in which the notion of the ADM transition is important is evaluation. My formulation of the relevant questions are as follows.

Can the effect of intermediate commands in a command list be observed by:

- the agent that instigated the change?
- another automated agent?
- the modeller? (who is also an agent)

Slade appears to be silent on these questions: the quotations given so far refer to evaluation of guards, and step 5 in Slade's ADM algorithm (Listing 2.2) states only "simulate parallel execution of all the command lists in the run set" and so is under-specified in this respect.

Beynon, however, informally describes the operation of the machine in many of the early sources. The representative quotation¹⁴ below, with my emphasis added, is taken from [Bey90, §2].

*... Each action is a sequence of instructions...
... On each machine cycle the guards associated with actions in A are evaluated in the context specified by the definitions in D. If there is no interference, those actions that are associated with true guards are then executed in parallel. Evaluation required in a redefinition... is performed in the same context as guard evaluation.*

The two emphasised parts of the quotation can be deemed to be in conflict: if evaluation is always performed in the same state as guard evaluation, then later commands in a command list cannot observe the state changes made by commands earlier in the list, and therefore command lists can be considered to be sets as well as sequences. However, there is no conflict, as long as a qualification is made to the effect that actions are sequences in respect of instantiation and deletion only, as described above under 'command sequencing'.

¹⁴Similar descriptions appear in [Bey88, p.8], [BSY89, p.3] and [BNOS90, p.4].

To aid thought on the matters of transitions and evaluation, I propose the use of the terms ‘major’ and ‘minor’ transitions¹⁵. In the ADM, the ‘transition’ considered so far, which is made up of sets of entire command lists, can be named a *major transition*. Each individual command within a command list forms a *minor transition*. A major transition moves the state from an initial state S to a resultant state S' . Minor transitions may produce intermediate S^* states between S and S' . The question I posed under the heading of ‘Evaluation’ above then becomes: can intermediate S^* states be observed?

I use the major/minor, S, S^*, S' terminology later in the thesis when talking of dependency-as-agency. Regarding the ADM, the second emphasised statement in the quotation from Beynon above can be read as: evaluation in a minor transition observes the S state formed by the last major transition, not an intermediate S^* state formed by a minor transition.

Slade’s `am` implementation follows this scheme, as can be seen from the experimental interaction shown in Listing 2.3 (which is slightly edited to reduce the size of the output). The entity `test` created in the listing contains two actions, which perform evaluation and change of the same variable in the two possible sequences:

```
b=|a|; a=2;
```

and

```
a=2; b=|a|;
```

Despite the differences in sequence ordering of the command lists, it can be seen that the value of variable `b` in the resultant S' state always takes the value of variable `a` as it was in the initial S state. Although the `am` implementation executes the commands in the command lists sequentially and in the provided order, the evaluations observe the initial S state. This is illustrated in Figure 2.5. As the intermediate S^* states (denoted in the figure by unfilled rectangles) are not observed, the sequential implementation can therefore be interpreted as acting in parallel, as shown at the bottom of the figure.

¹⁵A musical reader might imagine examples of these to be (`c=down // e=down // g=down`) and (`c=down // eb=down // g=down`) respectively, but they are not defined in this way here — please read on.

```

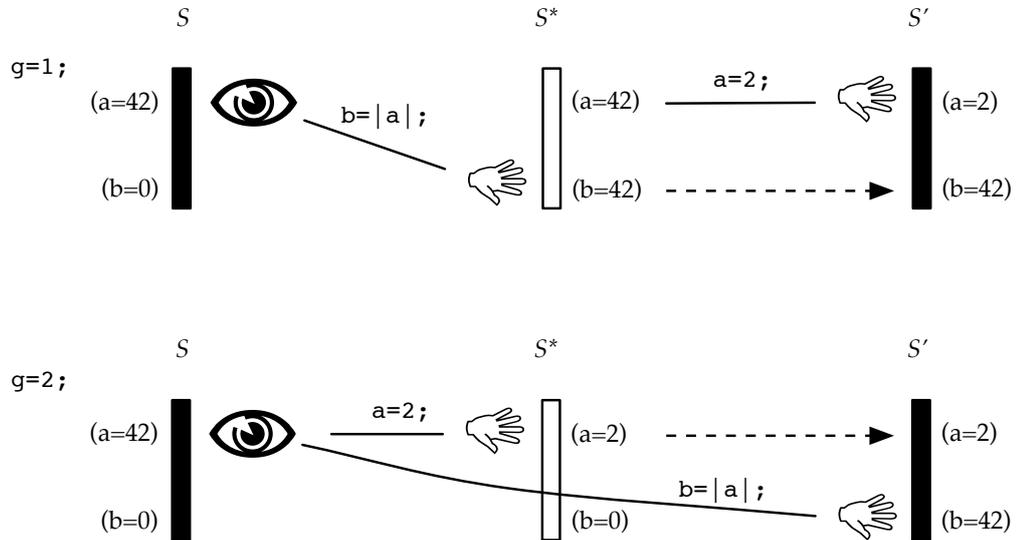
$ /dcs/emp/empubliс/bin/am-1.1
am-1.1> entity test() {
    definition
        a=0, b=0, g=0
    action
        g==1 print("g1") -> b=|a|; a=2; g=0,
        g==2 print("g2") -> a=2; b=|a|; g=0
}
am-1.1> test()
am-1.1> define a=42; define b=0;
am-1.1> define g=1;
am-1.1> l ds
Variable # 1: a = 42
Variable # 2: b = 0
Variable # 3: g = 1
am-1.1> start
g1
am-1.1> l ds
Variable # 1: a = 2
Variable # 2: b = 42
Variable # 3: g = 0
am-1.1> define a=42; define b=0;
am-1.1> define g=2;
am-1.1> l ds
Variable # 1: a = 42
Variable # 2: b = 0
Variable # 3: g = 2
am-1.1> start
g2
am-1.1> l ds
Variable # 1: a = 2
Variable # 2: b = 42
Variable # 3: g = 0
am-1.1>

```

(Note: User input is shown like this.)

Listing 2.3: An interaction with `am` demonstrating evaluation in S state

an sequential ADM
implementation



Parallel conception of
ADM execution

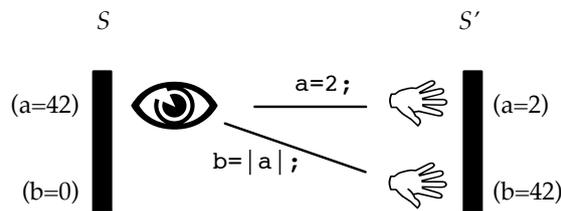


Figure 2.5: Sequential am implementation and parallel conception of the ADM

2.3.4 Divisible command lists and the ‘Authentic’ ADM (AADM)

Attempts have been made to use ‘divisible’ command lists in the ADM for simulation purposes, extending the use of command lists beyond the original purpose intended by Slade of supporting instantiation and deletion.

Y.P. Yung’s railway passenger example (shown in Figure 2.2 on p.51 and considered earlier in terms of parallel action in §2.3.1) gives one example. Yung’s LSD account includes the excerpt shown (slightly simplified) at the top of Figure 2.6. Yung translates this portion of the LSD account to the ADM script shown (again

```

agent passenger(...) {
  state pat[p], pos[p], alighting[p];
  oracle queuing[d], door_open[d];
  privilege alighting[p] ^ pos[p]==0 ^ door_open[d] ^ ¬queuing[d]
  → pos[p]=1; pat[p]=|at|; door_open[d]=false; pos[p]=2;
  ...
}

entity passenger(...) {
  definition
    pat[_p] = ...,
    pos[_p] = ...,
    alighting[_p] = ...,
    state[_p] = 0,
    ...
  action
    alighting[_p] && pos[_p]==0 && door_open[_d] && !queuing[_d]
      print("Passenger ",_p," alighting on platform")
      → pos[_p] = 1; state[_p] = 1; pat[_p] = |at|,
    state[_p]==1 && door_open[_d] && !queuing[_d]
      print("Passenger ",_p," closes door ",_d)
      → door_open[_d] = false; state[_p] = 2,
    state[_p]==2
      print("Passenger ",_p," leaves the station")
      → pos[_p] = 2; state[_p] = 0,
    ...
}

```

Figure 2.6: Excerpts from Y.P. Yung’s railway passenger LSD account and ADM script

slightly simplified) at the bottom of Figure 2.6.

Notice how Yung introduces an additional state variable, named `state`, and splits the single LSD command list into multiple ADM actions, in part guarded by the additional variable `state`. This technique is necessary in order to implement divisible command lists in the ADM. LSD command lists conceptually execute sequentially and the intermediate states are visible to other agents. ADM command lists however execute conceptually in parallel and the intermediate states are not visible to other entities. Yung has determined, when translating from LSD to the ADM, that the LSD command list shown should be translated to the three ADM actions shown, the two underlined LSD commands being synchronised (as described

am	<code>a = b;</code>	<i>(re)definition</i>
	<code>a = b ;</code>	<i>evaluation in S</i>
adm	<code>a is b;</code>	<i>(re)definition</i>
	<code>a = b;</code>	<i>evaluation in S</i>
adm2	<code>a is b;</code>	<i>(re)definition</i>
	<code>a = b;</code>	<i>evaluation in S*</i>
	<code>a = b ;</code>	<i>evaluation in S</i>

Figure 2.7: Syntax for redefinition and evaluation from three implementations of the ADM

in §2.3.1) and therefore being mapped to a single ADM action.

A separate, partial, attempt at using divisible command lists was made by Y.P. Yung in the second version of his ADM to Eden translator, `adm2`. Yung’s first translator implementation of the ADM, named `adm`, was used by Emma Davis in 1995 for modelling interaction within a classroom [Dav95]. Her report states:

The first difficulty I came across was that the lines of code within each action section were executed concurrently. While the action sections themselves should be executed concurrently, the code within them should be executed sequentially.

The operational semantics of Yung’s `adm` therefore initially matched that of Slade’s `am`. Yung solved Davis’s problem by introducing a second kind of evaluation distinguished by a different syntax, described in [Yun96, §6.1] and produced a new translator, named `adm2`. The relevant parts of the syntax of the three implementations is shown in Figure 2.7. Yung’s `adm` implementations deviate from the syntax given earlier in Figure 2.3 (p.56) as they use the `=` operator to denote evaluation and `is` to denote (re)definition, in order to ease translation to the Eden language. More important here are the varied operational interpretations. Figure 2.7 shows that `adm2` offers two types of evaluation.

Allowing evaluation in an intermediate S^* reveals the many possible command interleavings that are abstracted away by the ADM execution as described in the previous section. Yung [Yun96, §6.1] warns that “in a concurrent execution environment assigning variables in the context of execution, in principle, is dangerous; there is no guarantee what value you have assigned”. Use of such evaluation in the example shown in Figure 2.5 could lead to the variable b in the resultant state taking the value of 42 or 2, depending upon the particular interleaving used by the implementation.

Slade’s description of the ADM algorithm (Listing 2.2, p.69) and his associated observation to that effect that “command lists are treated as a single transition, no matter how many definitions they involve” indicate that he identified the ADM with a machine that executes parallel actions in an indivisible fashion. Yung’s railway animation demonstrates that such an interpretation of ADM execution is consistent with the animation of an LSD account, subject to grouping the sequence of commands in the LSD agents into indivisible segments, and ‘programming’ the ADM to execute these accordingly. This approach to simulation is what Slade would have recognised as ‘using the ADM to animate an LSD account’.

The correspondence between LSD and the ADM highlighted in Table 2.1 (p.59) commends a conceptually very different interpretation¹⁶ of ‘using the ADM to animate an LSD account’. In this approach to LSD animation, each LSD agent corresponds to an ADM entity and contributes a set of actions for which the command lists are to be conceived as sequences of atomic commands. For the purposes of simulating behaviour on the basis of an LSD account — bearing in mind the status of LSD actions as privileges rather than obligations for action — a mode of execution quite unlike that described by Slade in the ADM algorithm of Listing 2.2 is appropriate. I have specified this alternative “mode of execution” in Listing 2.4, which is based on Beynon’s informal accounts (e.g. [BACY94a, BACY94b]) of how the ADM was intended to be used to give operational meanings to an LSD account.

¹⁶In Slade’s terms, this would be seen as a misinterpretation, since it presumes that command lists that are divisible, and comprise sequences of atomic commands.

Listing 2.4 documents (seemingly for the first time in the EM literature) the initial conception of a framework for LSD animation by Beynon and Slade that led to the identification of the ADM. In Slade’s account of this research in [Sla90], the term ‘ADM’ refers to the ‘computational engine’ that is used to implement the parallel execution of indivisible commands specified at Step 12 in Listing 2.4. Unfortunately, in writing about the use of the ADM in LSD animation, Beynon has consistently (e.g. [BACY94a, BACY94b]) referred to the ADM as if its authentic mode of execution was as I have detailed it in Listing 2.4. This usage of the term ADM in connection with divisible command lists differs radically from the conception of the ADM introduced by Slade, where command lists are understood to be indivisible. On this basis, I propose that Beynon’s use of the term ‘ADM’ to refer to execution with divisible command lists be deprecated, and that the term ‘Authentic’ ADM (AADM) be adopted where the mode of execution specified in Listing 2.4 is presumed. It is to the AADM concept that Rungrattanaubol implicitly refers in the ‘Abstract Definitive Modelling framework’, “a conceptual framework for multi-agent construal” [Run02, p.83], introduced in her doctoral thesis [Run02, §3].

The AADM algorithm in Listing 2.4 combines a ‘reliable’ part for evaluating functional dependencies (a ‘machine’ component) and an ‘unreliable’ part for simulating agency (a ‘human’ component). Entities in a typical execution of the AADM are derived directly from state observables of agents in an LSD account (*cf.* Table 2.1, p.59). The AADM then serves as “an instrument for interpreting¹⁷ LSD”.

A transition in the AADM algorithm is formed from a set of single commands selected from command lists. Each command list is executed sequentially, as in the ADM. However, each command is itself atomic and the resulting intermediate S^* states (between the commands in a command list) are observable to other agents. The commands within the set of active command lists are interleaved non-deterministically.

In the context of Listing 2.4, a command is intended to denote an “atomic action” on the part of an agent. This would normally be represented by a redefinition or an entity invocation/deletion. In some circumstances, it may be appropriate for a command to take a composite form. For instance, as discussed in §2.3.1, a

¹⁷In both the human and machine senses.

In each step:

1. (The state is now S .)
2. For each action a :
 3. If the action a is currently executing and there is no command from a already in the run set (pending execution):
 4. Add the next command in action a to the run set.
 5. Else:
 6. Evaluate guard of a in state S .
 7. If guard of a is true:
 8. Add the first command in action a to the run set.
9. Check the run set for an invalid transition.
10. If the transition is invalid,
 11. Stop and ask the modeller to resolve the conflict before proceeding.
12. Select a subset of the commands from the run set and execute these, conceptually in parallel, making the transition to state S' .
13. (The state is now S' .)
14. Go to 1.

Notes on step 12:

- Selection of the subset of commands can be determined non-deterministically by the algorithm or determined by the ‘super-agent’ modeller.
- Due to the guarantee given by the invalid transition check, the run set contains no interference between commands.
- Commands can therefore be performed sequentially *or* in parallel.
- If commands are performed sequentially, the state will move through intermediate S^* states before it arrives at S' . Evaluations can be performed in the S or S^* states without any difference to the result as there is no interference between commands.

Listing 2.4: A proposed algorithm for ‘Authentic’ ADM (AADM) execution (Ward, after Beynon and Slade)

single agent may be deemed to perform several actions in parallel (“the pianist plays a chord”). An entity invocation may also involve the introduction of many redefinitions in parallel (“Meurig enters the office”). Both of these scenarios can be regarded as instances of “block redefinition”. Composite sequences of instructions may also be required in implementing commands. Slade’s use in the ADM of a sequential but atomic “command list” for instantiation and initialisation of an entity (*cf.* §2.3.3) is the prototypical example. More generally, depending on the nature of the definitive notations at the disposal of the modeller, it may not be possible to represent the transformation of a complex geometric object by a single redefinition. For instance, relocating a 3-dimensional object may presume parallel redefinition of many coordinates. Even where definitive notations to address such a requirement exist, the indivisibility of redefinition must be guaranteed in the implementation. More complex issues of atomicity are raised by transitions that involve what previous authors have described as “Higher Order Definition” (HOD). Topics relating to the above concerns are addressed throughout the thesis — see for example §3.1.2 on block redefinition and §4.3.7 and §5.3 on HODs. They are associated with the agenda of “Implementing Dependency on Digital Computers”.

The use of the word ‘select’ at step 12 reflects the idea that the non-determinism is potentially under the control of the modeller as “super-agent”. Non-determinism and super-agency are concepts discussed by Slade¹⁸ in [Sla90, §8.4] under the heading “Extensions to the adm”, but are absent from the detailed descriptions of the ADM itself, as the above subsections have discussed. In discussing how LSD accounts and ADM artefacts are related in EM, Beynon frequently alludes implicitly to three different modes of execution of the ADM. These correspond to three strategies for implementing the selection of commands to be performed at step 12:

- Manual selection of commands by the modeller;
- Automated selection of specific commands following some empirically plausible pattern (for example, using probabilistic techniques);
- Automating the selection of *all* commands other than those that are intended to represent actions on the part of human agents.

¹⁸Calling them asynchronous execution and ‘superuser’ respectively.

Which of these strategies is appropriate depends in general on the maturity of the modeller's understanding of agency as it develops during the modelling activity (*cf.* the reference to View 1, View 2 and View 3 agents in §2.1.1).

The AADM will not be considered further here, as it is most closely associated with the semantics of automated agency, and the primary concern of this thesis is the implementation of dependency.

2.4 The ADM and UNITY

UNITY ([CM88], on which much of the following description is based) is a theory of programming, comprising:

- a method for specification of programs;
- a method of reasoning about specifications;
- a method of developing programs with a proof that they meet their specification;
- a method of transforming programs to achieve high efficiency on the machines available for their execution.

UNITY generalises the theory of sequential programming [Dij76] for use with distributed and concurrent algorithms. Although some notational syntax is given in [CM88] for UNITY, UNITY is not considered a programming language by the authors — the notations are introduced in order to illustrate their computational model and proof system. UNITY stands for Unbounded Nondeterministic Iterative Transformations, describing a concurrent computational model that can be mapped to other computation models, including sequential ones.

UNITY programs consist of a declaration of variables, a specification of their initial values and a set of *multiple-assignment* statements. Figure 2.8 shows a selection from the BNF description of the UNITY notation. Syntactic units enclosed between { and } may be instantiated zero or more times.

<i>program</i>	::=	Program	<i>program-name</i>
		declare	<i>declare-section</i>
		always	<i>always-section</i>
		initially	<i>initially-section</i>
		assign	<i>assign-section</i>
		end	
<i>assign-section</i>	::=	<i>statement-list</i>	
<i>statement-list</i>	::=	<i>statement</i> { \square <i>statement</i> }	
<i>statement</i>	::=	<i>assignment-statement</i> <i>quantified-statement-list</i>	
<i>assignment-statement</i>	::=	<i>assignment-component</i> { <i>assignment-component</i> }	
<i>assignment-component</i>	::=	<i>enumerated-assignment</i> <i>quantified-assignment</i>	
<i>enumerated-assignment</i>	::=	<i>variable-list</i> := <i>expr-list</i>	
<i>variable-list</i>	::=	<i>variable</i> {, <i>variable</i> }	
<i>expr-list</i>	::=	<i>simple-expr-list</i> <i>conditional-expr-list</i>	
<i>simple-expr-list</i>	::=	<i>expr</i> {, <i>expr</i> }	
<i>conditional-expr-list</i>	::=	<i>simple-expr-list</i> if <i>boolean-expr</i> { \sim <i>simple-expr-list</i> if <i>boolean-expr</i> }	

Figure 2.8: Partial BNF for UNITY (summarised from [CM88, §2])

Some examples of *enumerated-assignments* are given in [CM88, §2.3.2]:

1. Exchange x, y .

$$x, y := y, x$$

2. Set x to the absolute value of y .

$$x := y \text{ if } y \geq 0 \sim -y \text{ if } y \leq 0$$

3. Add $A[i]$ into sum and increment i , provided i is less than N .

$$sum, i := sum + A[i], i + 1 \text{ if } i < N$$

4. Assign the smaller of $A[i]$ and $B[j]$ to $C[k]$ and increment k by 1; also increment i if $A[i] \leq B[j]$ and increment j if $B[j] \leq A[i]$.

$$\begin{aligned} C[k] &:= \min(A[i], B[j]) \\ \parallel \quad k &:= k + 1 \\ \parallel \quad i &:= i + 1 \quad \text{if } A[i] \leq B[j] \\ \parallel \quad j &:= j + 1 \quad \text{if } B[j] \leq A[i] \end{aligned}$$

or

$$\begin{aligned} C[k], i, j, k &:= \\ A[i], i + 1, j, k + 1 &\quad \text{if } A[i] < B[j] \sim \\ B[j], i, j + 1, k + 1 &\quad \text{if } A[i] > B[j] \sim \\ A[i], i + 1, j + 1, k + 1 &\quad \text{if } A[i] = B[j] \end{aligned}$$

An *assignment-component* can also be a *quantified-assignment*, allowing a finite number of instances of assignment to be created from the one specification. The brackets \langle and \rangle delineate quantification scope.

$$\begin{aligned} \textit{quantification} &::= \textit{variable-list} : \textit{boolean-expr} :: \\ \textit{quantified-statement-list} &::= \langle \parallel \textit{quantification statement-list} \rangle \\ \textit{quantified-assignment} &::= \langle \parallel \textit{quantification assignment-statement} \rangle \end{aligned}$$

Some examples of *quantified-assignments* are given in [CM88, §2.3.3]:

Given arrays $A[0..N]$ and $B[0..N]$ of integers, assign $\max(A[i], B[i])$ to $A[i]$, for all $i, 0 \leq i \leq N$.

$$\langle \parallel i : 0 \leq i \leq N :: A[i] := \max(A[i], B[i]) \rangle$$

or

$$\langle \parallel i : 0 \leq i \leq N :: A[i] := B[i] \text{ if } A[i] < B[i] \rangle$$

or

$$\langle \parallel i : 0 \leq i \leq N \wedge A[i] < B[i] :: A[i] := B[i] \rangle$$

Program execution in the UNITY model starts from any state satisfying the initial condition and goes on forever. In each step of execution, some assignment statement is selected non-deterministically and executed. The non-deterministic selection is constrained by the “fairness rule”: every statement is selected infinitely often.

Sequential control flow is therefore absent in the UNITY programming model. The passage below [CM88, §1.2.2] explains why.

The notion of sequential control flow is pervasive in computing. Turing Machines and von Neumann computers are examples of sequential devices. Flow charts and early programming languages were based on the sequential flow of control. Structured programming retained sequential control flow and advocated problem decomposition based on the sequencing of tasks. The prominence of sequential control flow is partly due to historical reasons. Early computing devices and programs were understood by simulating their executions sequentially. Many of the things we use daily, such as recipes and instructions for filling out forms, are sequential; this may have influenced programming languages and the abstractions used in program design.

The introduction of co-routines was an indication that some programs are better understood through abstractions unrelated to control flow. A program structured as a set of processes is a further refinement: it admits multiple sequential flows of control. However, processes are viewed as *sequential* entities — note the titles of two classic papers in this area, “Cooperating Sequential Processes” in Dijkstra [Dij68] and “Communicating Sequential Processes” in Hoare [Hoa78]. This suggests that sequential programming is the norm, and parallelism, the exception.

Control flow is not a unifying concept. Programs for different architectures employ different forms of control flow. Program design at early stages should not be based on considerations of control flow, which is a later concern. It is easier to restrict the flow of control in a program having few restrictions than to remove unnecessary restrictions from a program having too many.

The issue of control flow has clouded several other issues. Let us review one. Modularity is generally accepted as a good thing. What is a module? It implements a set of related concerns, it has a clean, narrow interface, and the states of a system when control flows into and out of the module are specified succinctly. Now a clean, narrow interface is one issue and control flow into and out of a module is another. Why not separate them? In our program model, we retain the concept of a module as a part of a program that implements a set of related concerns. Yet we have no notion of control flow into and out of a module. Divorcing control flow from module construction results in an unconventional view of modules and programming — though a useful one, we believe, for the development of parallel programs.

Control flow is however specified in UNITY: not in the programming model, but in the mapping of a UNITY program to an architecture. A mapping specifies how to partition the set of statements of a program amongst processors, how to

partition the set of variables among memories and channels, and the control flow for each processor. The mappings to synchronous architectures considered in [CM88, §4.2.3] are “particularly simple... In the mapping employed in this book, exactly one statement of the program is executed at a time, regardless of the number of processors available”.

There are many similarities between the UNITY model and the ADM. In an intriguing parallel with the motivation behind the work presented in this thesis, “some of the initial motivation for UNITY came from difficulties encountered in using spreadsheets, a notation that has not received much attention from the computing-sciences community” [CM88, p.19].

Conditional assignments in UNITY correspond closely to guarded command lists in ADM scripts. Compare the ADM script fragment:

$$y \geq 0 \rightarrow x = y,$$

$$y \leq 0 \rightarrow x = -y$$

to the UNITY (from [CM88, p.25])

$$x := y \text{ if } y \geq 0 \sim -y \text{ if } y \leq 0$$

for example.

Chandy and Misra [CM88, §22.7.3] compare UNITY’s conditional assignments to guarded commands as defined in [Dij76]. Aside from the syntactical differences shown in the example above (mainly relating to whether the conditions/guards or actions are emphasised), there is also a semantic distinction, which the following passage explains:

The semantic differences between guarded commands and UNITY have to do with fairness. In the guarded command theory, a statement is selected for execution only if its guard is *true* (or “enabled”). If an arbitrary statement is selected for execution, it is possible that a statement whose guard is *false* is chosen forever — because there is no fairness constraint — and then there is no progress of computation in this case. Therefore the notion of a guard is crucial; only the statements with enabled guards are eligible for execution.

In UNITY, the fairness rule obviates the need for guards. A statement whose execution does not change the program state may be selected for execution; it can, however, be executed only a finite number of times. Therefore a statement whose execution changes program state — if such a statement exists — is selected eventually for execution. This is how progress of computation is guaranteed in UNITY. There is no notion of guards or enabling, and this has simplified the proof theory.

The motivation for the fairness rule in UNITY is that of simplifying reasoning about progress properties [CM88, p.489]. The radical form of non-determinacy conceived in the AADM is also a departure from the traditional operational interpretation of guarded commands. It is motivated by a desire to match simulation behaviour to behaviour as it is observed in the referent.

UNITY programs can contain an “always-section” which is similar to the definitions section in an ADM script. The always-section in UNITY “is used to define certain program variables as functions of other variables” [CM88, §2.7]. The syntax of the always-section follows that of the “initially-section”, which follows that of the “assign-section”, except that in assignments ‘:=’ is replaced by ‘=’. A variable appearing on the left side of an equation in the always-section is called a “transparent” variable.

A transparent variable is a function of nontransparent variables and hence does not appear on the left side of any initialization or assignment, though it may appear on the right. A transparent variable may also appear on the right side of an equation.

To ensure that each transparent variable is a well-defined function of nontransparent variables, UNITY places restrictions on transparent variables. The restrictions are the same as for variables in the initially-section. The equations in the initially-section and transparent variables should not be circular. The set of equations must be “proper”, defined as [CM88, §2.5]:

1. a variable appears at most once on the left side of an equation,
2. there exists an ordering of the equations such that any variable in a quantification is either a bound variable or a variable that appears on the left side of an equation earlier in the ordering (ensuring that the program can be “compiled”), and
3. there exists an ordering of all equations after quantified equations have been expanded such that any variable appearing on the right side of an equation, or in a subscript, appears on the left side of an equation earlier in the ordering (ensuring that the initial values are well defined).

An example given ([CM88, §2.7]) for a transparent variable is

$$ne = nf + nm$$

where nf , nm , ne denote the number of female employees, male employees and employees respectively.

The following passage ([CM88, §2.7]) motivates the always-section:

... First, it is simpler to reason with an always-section because it defines a set of invariants of the program. . . These invariants are in a particularly nice form, a set of equations. If a program contains only transparent variables, it can be regarded purely as a set of equations, and it is usually easier to understand such programs. Second, it is convenient to regard a transparent variable merely as a macro whose definition can be substituted for its occurrence anywhere in the program. The term transparent comes from this property of *referential transparency*. Third, efficient implementations of transparent variables are possible: Evaluation of a transparent variable can be deferred until it is accessed or until some of the variables in its definition change value.

Transparent variables in UNITY are thus similar to definitions in the ADM. However, the UNITY always-section cannot be changed at run-time: the always-section specifies invariants, or fixed constraints. This assists with a proof of correctness. The ADM in comparison could be thought of as allowing a sequence of “run-times”, the sequence being formed through modification of definitions. The “transparent” nature of the definitions makes analysis of the static state possible: definitions provide meaningful state.

Finally, Chandy and Misra [CM88, Epilog] propose what might be called a methodology for developing UNITY programs.

The change in a programmer’s primary concern, from a specific machine — i.e., one in Taylor Hall — to a generic machine — i.e., a FORTRAN machine — was the first in a series of steps towards increasing abstraction. . . But then computer architects began introducing strange machines. They demolished the comforting cocoon of conformity.

The UNITY response to the challenge of novel machines is to propose yet another answer to the question What is programming? Our answer is a logical progression to the answers given by our programming forebears. Once again we generalize our view of programming.

Programming is the art of making a series of stepwise refinements of specifications. We begin with a large space of potential solutions, each refinement rules out some solutions, and we end with a small number of “good” concrete solutions. We expect that the decisions we make early in our designs are appropriate for *all* architectures. As design proceeds and the target architectures are defined more narrowly, our decisions are more appropriate for smaller sets of architectures. This approach is not unlike that made by programmers who, when opting for one PASCAL data structure rather than another, expect their decisions to be appropriate for *all* machines on which their programs run. In the final stages of design, programmers may code a few subroutines in assembly language to exploit features of a target machine. But programmers know that it is not cost effective to *begin* design by programming in the assembly language of all machines on which their programs may be required to execute.

The core of programming is not concerned with a specific architecture, a specific operating system, or a specific programming language any more than it is

concerned with the machine in the basement. The core of programming is a theory that allows programmers to make a series of design decisions. This book has attempted to present such a theory.

The above discussion of UNITY is of interest because of several points of contact with the concepts behind the ADM and its use in animating LSD. The ADM that Slade identifies can be seen as an “abstract computational model” that could in principle be the basis for a formal specification of behaviour in much the same spirit as UNITY. However, the distinctive perspective that is characteristic of EM is only apparent when the AADM is considered. The simulation activity specified in Listing 2.4 (p.79), which combines manual and automated interaction, is a more closely integrated concrete and situated activity than is normally associated with “a refinement methodology”. The perception of indivisibility and the engineering of artefacts through experimental interaction play an essential role in this activity. For more details, and an elaboration of the theme, the interested reader may consult Rungrattanaubol [Run02]. In the context of this thesis, the most significant implication is that the implementation of dependency and agency in EM tools cannot be treated as an abstract programming exercise. The way in which state is perceived and engineered is crucial to the successful implementation of AADM simulation.

2.5 Background/Sources

The Abstract Definitive Machine has been written about in many papers and theses.

The papers can be categorised into two approximate themes: parallel programming and design and modelling. Parallel programming was an early theme for publications containing material on the ADM. The first publication is [Bey88], which gives the first treatment of a block-moving example, followed by [BSY89] which describes a simulation of a systolic array algorithm, [Bey90] which gives the fullest treatment of the block-moving example and [BNOS90] which discusses the relationship of LSD accounts and the ADM in the context of modelling a telephone exchange.

In the design and modelling theme papers category, [Bey89c] was the first journal publication to present the block-moving example (an example which straddles both thematic categories) as an ADM ‘program’ along with a graphical representation

in DoNaLD. [Bey89a] is concerned with definitive programming in design, [BRY90] presents an LSD model of a cat flap, [BY92] considers modelling and simulation of activity at a railway station as an LSD account and an ADM ‘program’, [BC93] envisages some extensions for specifying ADM entities and finally [BACY94a] contains a figure representing the ADM machine model. The ADM also gets brief mention in many other papers, including [BY90, BACY94b, BACY94c, Bey94, BY94, BC95, Bey97, Bey99].

The most detailed source about the ADM ideas is Slade’s MSc thesis [Sla90], which is quoted extensively in this chapter. It contains the original treatment of the block-moving example and the telephone exchange model and also describes the first implementation `am`.

Y.P. Yung’s PhD thesis [Yun93], [Yun96] (a post-doctoral report) and P-H. Sun’s PhD thesis [Sun99] present ADM ‘program’ to Eden translators named `adm`, `adm2` and `adm3`, which are not the main focus of this thesis chapter. Additionally, [Yun93, §2.1] describes the Definitive State Transition (DST) model, similar to the ADM machine model. Rungrattanaubol [Run02] reviews the area, somewhat confusingly reusing (§3.1) the upper case acronym ADM for Abstract Definitive Modelling (lower case `adm` is the Abstract Definitive Machine). Heron [Her02] shows how to convert an ADM ‘program’ into a JaM2 script [Car99]. Wong [Won03, §2.2] describes the Definitive Modelling Framework (DMF), which is also similar to the ADM model.

A reader of the sources needs to be aware that sometimes the source is describing LSD, sometimes ADM the machine model, or sometimes an implementation of the ADM, which might itself be a full machine implementation or a translator into some other language. Often, the focus is not explicitly declared, and so clues must be obtained from the context. Table 2.1 (p.59) assists in distinguishing LSD and ADM terminology. Distinguishing discussion of the ADM from that about an implementation is sometimes more problematic. Slade [Sla90] uses mathematical symbols (e.g. \wedge) when describing the ADM machine model and ADM ‘programs’ in the abstract but keyboard symbols (e.g. `&&`) when describing the `am` implementation.

My scholarship on the ADM literature has exposed a significant misunderstanding that has had an unhelpful influence over the comprehension and practical development of the ADM since Slade’s first implementation of `am`. It is apparent in

retrospect that the embryonic nature of the `am` implementation and confusion surrounding the authentic mode of execution of the ADM has inhibited the development of good tools to support EM development of concurrent systems. In particular, none of the existing EM tools combines manual and automated interaction, and divisible and indivisible execution of command lists, in a way that does justice to Beynon and Slade's original conception for LSD animation as I have identified it in this chapter. It is also clear that a successful implementation of a tool to support this conception will involve a more radical kind of revision of the `am` than has been envisaged to date; unlike `adm`, `adm2` and `adm3`, which are essentially rooted in automating atomic transitions, it will need to take full account of the partially-automated mechanism I have documented in Listing 2.4 (p.79). Some of the key technical issues to be addressed in developing such an implementation form the subject of the rest of my thesis.

2.A Using the *am* implementation

Listing 2.5 is a script for *am* based on a version by Y.P. Yung [BRY90] that models a cat, a man and a cat flap with a four-way lock. Notice, for example, that the man entity will set the cat flap lock to position 0 approximately once every 60 transitions, as long as the flap at angle 0 and the lock is not already at position 0.

```
entity flap() {
  definition
    lflap = 5,                # the length of the flap
    angle = 0,                # inclination of the flap
    switch = true,           # whether the electronic lock is
                             # operating
    fourWayLock = 0,         # 1: cat can go out only, 2: in only
                             # 0: no restriction, 4: no access
    Radius = 10,             # electronic lock detector range
    elecLock = (pos > Radius || pos < -Radius) && switch,
    canPushOut = angle != 0 || (!elecLock && (fourWayLock == 0 ||
        fourWayLock == 1)),
    canPushIn = angle != 0 || (!elecLock && (fourWayLock == 0 ||
        fourWayLock == 2))

  action
    pushOut && canPushOut
    print("Angle becomes ", angle+1)
    -> angle = |angle| + 1,
    pushIn && canPushIn
    print("Angle becomes ", angle-1)
    -> angle = |angle| - 1,
    !pushOut && !pushIn && angle > 0
    print("Angle becomes ", angle-1)
    -> angle = |angle| - 1,
    !pushOut && !pushIn && angle < 0
    print("Angle becomes ", angle+1)
    -> angle = |angle| + 1
}
}
```

```

entity man() {

definition
    actnow = 0

action
    actnow == 1 && switch == false
print("Switch on")
    -> switch = true,
    actnow == 11 && switch == true
print("Switch off")
    -> switch = false,
    actnow == 21 && angle == 0 && fourWayLock != 0
print("Four way lock is set to 0")
    -> fourWayLock = 0,
    actnow == 31 && angle == 0 && fourWayLock != 1
print("Four way lock is set to 1")
    -> fourWayLock = 1,
    actnow == 41 && angle == 0 && fourWayLock != 2
print("Four way lock is set to 2")
    -> fourWayLock = 2,
    actnow == 51 && angle == 0 && fourWayLock != 3
print("Four way lock is set to 3")
    -> fourWayLock = 3,
    true -> actnow = |rand(60)|

}

entity cat() {

definition
    height = 2,                # height of the cat
    pos = 2,                   # >0: outside the house, <0: inside
    obstruct = 0,
    intention = -1,           # 1: going out, -1 coming in,
                                # 0 stay put

    pushOut = intention > 0 && obstruct,
    pushIn = intention < 0 && obstruct

action
    intention > 0 && !obstruct
print("Cat moves outward")
    -> pos = |pos| + 1,
    intention < 0 && !obstruct
print("Cat moves inward")
    -> pos = |pos| - 1

}

```

Listing 2.5: Y.P. Yung's *am* cat flap script

Interaction with *am* in use is through a text interface only. Listing 2.6 shows an interaction with *am*¹⁹ that uses the script in Listing 2.5. User input is shown like this .

```

$ cd ~empub/public/projects/catflapYung1994
$ cat flap-noinstantiate.am - | /dcs/emp/empub/public/bin/am-1.1
am-1.1> compiling flap()
am-1.1> compiling man()
am-1.1> compiling cat()
am-1.1> l en                                     # list entity descriptions (P)

      ENTITY LIST
      *****
entity flap() {      (0 parameters)
...

entity cat() {      (0 parameters)
DEFINITION
  height = 2,
  pos = 2,
  obstruct = 0,
  intention = -1,
  pushOut = intention>0&&obstruct,
  pushIn = intention<0&&obstruct
ACTION
  intention>0&&!obstruct print("Cat moves outward") ->
    pos = |pos|+1,

  intention<0&&!obstruct print("Cat moves inward") ->
    pos = |pos|-1
}
0 instances
      END OF ENTITY LIST
      *****
am-1.1> l in                                     # list instances

INSTANCES
*****
      END OF INSTANCES
am-1.1> l ds                                     # list definition store (D)

DEFINITION STORE
*****
      END OF DEFINITION STORE
*****

```

¹⁹Note: on line 2, *cat* is a standard UNIX command, not a feline.

```

am-1.1> l as # list action store (A)
ACTION STORE
*****
END OF ACTION STORE
*****

am-1.1> cat () # instantiate new cat entity
instantiating cat
am-1.1> l in
INSTANCES
*****
cat ()
END OF INSTANCES
am-1.1> l ds
DEFINITION STORE
*****
Variable # 1: height = 2
Variable # 2: pos = 2
Variable # 3: obstruct = 0
Variable # 4: intention = -1
Variable # 5: pushOut = intention>0&&obstruct
Variable # 6: pushIn = intention<0&&obstruct
END OF DEFINITION STORE
*****

am-1.1> l as
ACTION STORE
*****
Action # 1: intention>0&&!obstruct print("Cat moves outward") ->
pos = |pos|+1
Action # 2: intention<0&&!obstruct print("Cat moves inward") ->
pos = |pos|-1
END OF ACTION STORE
*****

am-1.1> ?(pos) # show current definition and value
pos is defined as 2
pos evaluates to 2
am-1.1> set iterations = 4 # limit to 4 major transitions only
am-1.1> start # start the machine executing
starting simulation
#
Cat moves inward
#
Cat moves inward
#
Cat moves inward
#
Cat moves inward
* 4 iterations successfully completed

```

```

am-1.1> ?(pos) # cat is now inside the house
pos is defined as -1-1
pos evaluates to -2

# cat now wants to go out
am-1.1> define intention = 1;
defining intention
am-1.1> flap() # instantiate the cat flap entity
instantiating flap
am-1.1> define obstruct = pos == 0 && angle < 3;
# ideally this is defined in terms of
# lflap * tan(angle), pos and height,
# but am does not have tan()

defining obstruct
am-1.1> l ds # we now have definitions from flap

      DEFINITION STORE
      *****
Variable # 1: height = 2
Variable # 2: pos = -1-1
Variable # 3: obstruct = pos==0&&angle<3
Variable # 4: intention = 1
Variable # 5: pushOut = intention>0&&obstruct
Variable # 6: pushIn = intention<0&&obstruct
Variable # 7: lflap = 5
Variable # 8: angle = 0
Variable # 9: switch = TRUE
Variable # 10: fourWayLock = 0
Variable # 11: Radius = 10
Variable # 12: elecLock = (pos>Radius||pos<-Radius)&&switch
Variable # 13: canPushOut = angle!=0||(!elecLock&&
      (fourWayLock==0||fourWayLock==1))
Variable # 14: canPushIn = angle!=0||(!elecLock&&
      (fourWayLock==0||fourWayLock==2))
      END OF DEFINITION STORE
      *****
am-1.1> set iterations = 8
am-1.1> start # output from major transitions
# are separated by '#' characters.
# Note the flap descends in parallel
# with the cat motion

starting simulation
#
  Cat moves outward
#
  Cat moves outward
#
  Angle becomes 1
#
  Angle becomes 2
#

```

```
Angle becomes 3
#
Cat moves outward
Angle becomes 2
#
Cat moves outward
Angle becomes 1
#
Cat moves outward
Angle becomes 0
* 8 iterations successfully completed
am-1.1> man()
instantiating man
am-1.1> start # man randomly acts on the flap
starting simulation
#
Cat moves outward
#
Cat moves outward
#
Cat moves outward
Four way lock is set to 1
#
Cat moves outward
#
Cat moves outward
#
Cat moves outward
#
Cat moves outward
Switch off
#
Cat moves outward
* 8 iterations successfully completed
am-1.1> ^D # exit am
$
```

Listing 2.6: An interaction with *am* and Y.P. Yung's cat flap

Chapter 3

The Definitive Assembly Maintainer (DAM) machine

Richard Cartwright's Definitive Assembly Maintainer Machine (DAM machine) is "a dependency maintenance tool written in assembly code that maintains indivisible relationships between words in an area of computer RAM store" [Car99, §5]. It is an example of a definitive system implementation for 1-agent modelling that uses evaluation/storage strategy 2, evaluate-at-redefinition (see §2.2.1). Partly as a result of this basis, its design gives much prominence to dependency and little to agency. It is based on Cartwright's "Dependency Maintainer Model" (DMM), [Car99, §4], which is the most explicitly detailed specification of an abstract definitive machine currently in existence. This chapter first briefly examines the DMM in order to get a top-down perspective of the DAM machine, similar to the approach taken by Cartwright, but placed in the context of this thesis with other definitive systems.

Taking the complementary bottom-up approach, the machine itself and a surrounding interactive application environment named !Donald are then studied. The environment is extended to allow experimentation with a bottom-up style of use, involving novel interactions with a form of "definitive assembler".

The bottom-up approach reveals two problems that can be considered at the abstract level. Firstly, a problem involving references encountered when attempting to use dependency in a symbol table gives a good example of higher-order dependency and the associated difficulties this poses for implementation. Secondly, Cartwright's

DMM and DAM machine design abstract away the *location* of maintained machine words. This is the standard convention in high-level languages. It is also a convention adopted in all existing work on definitive notations and systems, but it is shown that this abstraction causes problems when using dependency on data items whose representation is larger than one word or of variable size.

Finally a case study (constructed bottom-up) demonstrates proof of concept for configuring the DAM machine as a definitive text editor, where the question “why is that pixel white?” can be answered with reference to indivisible relationships within the store, rather than by reference to a previous write action of some unconstrained agent.

The primary source on the DAM machine and the DMM is Cartwright’s PhD thesis [Car99], which describes the application of the DAM machine to two areas. Firstly, [Car99, §5.5] describes application of the DAM machine to visualising of a geometrical shape from its function representation [PASS95], the state of each pixel being (indirectly) determined by a definition. Secondly, [Car99, §5.4] cites Allderidge [All97], an undergraduate final year project report that describes !Donald, an implementation of the DoNaLD notation that uses the DAM machine to perform definition maintenance. Cartwright [Car99, §6] goes on to describe a Java implementation of the ideas described here, called the Java Maintainer Machine Application Programming Interface (JaM Machine API). JaM has recently been developed into its second version as part of Cartwright’s continuing work in the BBC’s Research and Development department. JaM is not discussed in this thesis as the underlying operational model is very similar to the DAM machine.

The DAM machine has fewer associated secondary source publications than the ADM, being cited in just two papers. The 1996 paper by Gehring *et al* [GYC+96] was written as implementation was progressing, and contains some preliminary statements about the envisaged machine. A second, 1998, paper by Allderidge *et al* [ABCY98] contains two sections about the DAM machine that are derived from Cartwright, along with some discussion of strategies for translation of definitive scripts into procedural programs.

In the first section below, the mathematical model presented in [Car99, §4] is reviewed, then the ‘BRA’ algorithm that operates in the domain specified by the

mathematical model. Subsequent sections move on to the implementation of the DAM machine, the !Donald application and the extensions and insights described above. Sections 3.1, 3.2, 3.3 take the form of a review of the DAM machine, incorporating a critical evaluation of Cartwright’s account [Car99] and a clarification and exposition of details omitted from [Car99] that are essential for understanding my technical extensions and experimental work, as discussed in sections 3.4 and 3.5. The script digraph concept used in this chapter and throughout the thesis is described in an appendix to this chapter, §3.A (p.178).

3.1 The DMM and the BRA

3.1.1 State and the DMM

In [Car99, §4], Cartwright refers to the mathematical model he presents as “The Dependency Maintainer Model”. It is not the only possible model of dependency maintenance — this thesis presents several others, so in this thesis, we shall use the acronym DMM to denote the particular model of dependency maintenance described in [Car99].

The DMM is an attempt to formalise various properties of scripts written in a simple “low-level” definitive notation (denoted here by the acronym LLDN¹). LLDN was constructed by “examining scripts of dependencies and extracting from these common abstractions that can be used to reason about them” [Car99, p.105]. LLDN is not explicitly defined by Cartwright. In my analysis, LLDN has the following characteristics:

- a script is a finite set of definitions;
- a definition is an association of a value with an identity²;

¹Neither DMM or LLDN are abbreviations used in [Car99]: they are introduced here in order to distinguish these particular concepts from models of dependency maintenance in general and definitive notations in general respectively.

²Here I use identity rather than identifier as this identity is a numeric address.

Identity	Function	Arguments	Value
α	<i>times</i>	$(\alpha + 5, \alpha + 3)$	48
$\alpha + 1$	<i>power</i>	$(\alpha + 4, \alpha + 6)$	8
$\alpha + 2$	<i>value₄</i>	$()$	4
$\alpha + 3$	<i>double</i>	$(\alpha + 4)$	4
$\alpha + 4$	<i>value₂</i>	$()$	2
$\alpha + 5$	<i>add</i>	$(\alpha + 1, \alpha + 2)$	12
$\alpha + 6$	<i>value₃</i>	$()$	3

Table 3.1: An example script in LLDN

- values are taken from the infinite set of integers, \mathbb{Z} , which forms the only data type;
- identities are integer values in the contiguous range $[\alpha, \beta - 1]$;
- the current value associated with a particular identity is defined by the application of a single function to a (potentially unbounded) sequence of arguments;
- function arguments are values associated with identities;
- many functions are potentially available from the set of functions \mathcal{F} ;
- each function $f \in \mathcal{F}$ maps a sequence of integer values (the arguments) to a single integer value (the result).

An example of an LLDN “script” is given in Table 3.1 (which is derived from [Car99, Table 4.3]). The word script is often used to mean a *sequence* of lines in a text file: as in, for example, a shell script. The term “definitive script” in current usage usually implies the sequential meaning. It is sometimes drawn abstractly as a rectangle containing horizontal lines, implying a sequence of definitions. However, what I mean by an LLDN “script” is formed from a *set* of definitions. Hence, Table 3.1 is to be interpreted as a relation, like a database table, where each Identity (the primary key) is associated with a Function, Arguments and a Value.

Cartwright [Car99, §4.2.1] describes a four stage process which can transform “any script”³ [Car99, p.105] into a form representable in LLDN:

1. Every function in the script is replaced by a function in \mathcal{F} . This implies mapping all types available in the definitive notation to \mathbb{Z} . For example, the boolean definitions:

```
a is not(b); b is true;
```

might be replaced by:

```
a is boolnot(b); b is 1;
```

where the function `boolnot` : $\mathbb{Z} \rightarrow \mathbb{Z}$ maps i to $1 - i$. The DMM assumes that the data type used to represent \mathbb{Z} is of unbounded size, or is at least large enough to be able to represent the values required.

2. Function arguments (which may be sub expressions or literal values) are replaced by references to new definitions. For example,

```
a is boolnot(booland(b,1));
```

might be replaced by:

```
a is boolnot(x); x is booland(b,y); y is 1;
```

3. Now the total number of definitions is as it will be in LLDN form, so in stage three, variable names are replaced by integer identities. In the previous example, `a`, `x`, `y` and `b` might be replaced by the identities α through $\alpha + 3$ respectively. Note that the integer mapping is arbitrary: the script is still a set of definitions and adjacency of identities has no meaning.

4. Literal values are replaced by special “value functions” that take no arguments and return the value required. For example,

```
y is 23;
```

might be replaced by:

```
y is value23();
```

³But we shall see in §3.4.2, §3.4.3 and §3.5 that scripts using HOD or lists cannot be transformed into LLDN.

Cartwright [Car99, §4.2.2] explains that the state of an LLDN script is then characterised by a mapping of each of the identities in the current set to:

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

Formally, a transformed script \mathcal{S} can be represented by a DMM $\mathcal{M}_{\mathcal{S}}$, given by a 4-tuple:

$$\mathcal{M}_{\mathcal{S}} = (A, F, D, V)$$

where:

- A is the range $[\alpha, \beta - 1]$ of reference numbers for the script \mathcal{S} (the total number of definitions being equal to $\beta - \alpha$);
- F is a mapping $F : A \rightarrow \mathcal{F}$, associating a function from \mathcal{F} with every reference number in A ;
- D is a mapping $D : A \rightarrow A^*$, associating a sequence of arguments A^* (each argument itself being a reference to an identity) with every reference number in A ;
- V is a mapping $V : A \rightarrow \mathbb{Z}$, associating an integer value from \mathbb{Z} with every reference number A .

Having established these conventions, Cartwright goes on to formally define the function *lookup* (i.e. the association between reference number and value), the concept of *up_to_date* (i.e. that values are consistent with their dependencies), the presence of *cyclic dependency* (i.e. when a reference in the model directly or indirectly depends on itself), *dependencies* and *dependents* (in this thesis named sources and targets respectively, following [Yun90]), and the *stable state* (i.e. the DMM is up_to_date and no cyclic dependency is present).

3.1.2 Transitions and the block redefinition algorithm (BRA)

Thus far we have described the aspects of state in the DMM. Cartwright [Car99, §4.2.4] explains that a *transition* from a stable state:

$$\mathcal{M}_S = (A, F, D, V)$$

to another stable state:

$$\mathcal{M}_{S'} = (A', F', D', V')$$

can be described by a mapping K , determined by a subset of the set of all possible redefinitions, each of which (re)associates a function and a sequence of arguments with a reference number:

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

The new range of reference numbers $A' = [\alpha', \beta' - 1]$ may be extended from the original $A = [\alpha, \beta - 1]$ as new references introduced in K (on the left and/or right hand sides) may extend the range. It is not possible to remove definitions and hence reduce the range through a redefinition in K [Car99, p.115]: Cartwright does not consider undefined values or removal of definitions.

For a given block of redefinitions K , the set of references whose value is to be updated is $U(K)$ [Car99, p.118] — in the vocabulary of this thesis, the set of all recursive targets of K (see Appendix §3.A, p.178). Once K has been shown to meet two criteria described below, the values of all references in U should be updated in order to perform the state transition from \mathcal{M}_S to $\mathcal{M}_{S'}$. This kind of transition, which is only allowable if the two criteria are first met, is termed a *protected_update* [Car99, p.118]:

$$\textit{protected_update}(\mathcal{M}_S, K) = \begin{cases} \mathcal{M}_S & \text{if } \textit{cyclic}((A', F', D', V)) \\ & \text{or } \neg \textit{suitable}(K) \\ (A', F', D', V') & \text{otherwise} \end{cases}$$

The two criteria are that the new state must not contain cyclic dependency and that K must be *suitable* (Cartwright's term), that is to say, that for every new

reference in K , there is a new definition, as undefined values are not allowable in the DMM. The *protected_update* operation:

$$\mathcal{M}_{S'} = \textit{protected_update}(\mathcal{M}_S, K)$$

therefore encompasses some, but not all features of Slade's characterisation of a valid transition, described in the earlier §2.3.2. In Cartwright's terms, a valid transition is one where given \mathcal{M}_S and K , there is only one possible resulting $\mathcal{M}_{S'}$.

The amount of computation required to make the transition from \mathcal{M}_S to $\mathcal{M}_{S'}$ is the sum of the computation required to update each element in $U(K)$. Predicting the total time taken to make the transition is possible, in principle. The major transition \mathcal{M}_S to $\mathcal{M}_{S'}$ is formed from minor transitions. Each minor transition is formed by the update of the value of one definition, involving reading the arguments from store, executing the appropriate operator (a function in \mathcal{F}) and writing the result back into store. It should be possible to estimate the time required for each operator — this could be done by close examination of the machine code, summing the time required for each instruction. This measurement is complicated by the presence of caches on modern systems. The timing of some operators may also be dependent upon the data input. However, the effort of estimation would be repaid if the results were reused across many scripts. This is possible as it is the time required to evaluate each available operator that is being estimated, rather than the time required to evaluate each individual script⁴. Given the timings required for each minor transition, on a sequentially executing machine, the time for the major transition is then the sum of the individual minor transition times. On a parallel machine, some knowledge of the hardware and mapping of minor transitions to processors would be required in order to characterise the major transition. This problem is similar to that solved by the CHIP³S toolset developed at Warwick [PKNA95], which is now named PACE and is being applied to Grid computing environments [CJS⁺02].

A long-term goal for research in this area would be to derive a real-time characterisation of the performance of a dependency maintainer so comprehensive that it could be used to calculate the time for a major transition⁵ and compare this with

⁴This may be contrasted with the estimation of the performance of a conventional program (benchmarking), where e.g. the number of iterations of a loop may be data-dependent.

⁵Any transition or a particular restricted set of specific transitions.

real-time deadlines automatically. The technical issues to be addressed in such characterisation include estimation of evaluation times for the whole variety of operators on a specific platform. This is a major engineering task beyond the scope of this thesis. A simpler characterisation of performance, based on estimating the number of element updates, captures much that is useful.

Cartwright [Car99, p.125] showed that, if updates are appropriately scheduled, the number of updates associated with a single redefinition need not exceed the total number of definitions in the script.

In the case where the mapping K is determined by a set that contains more than one redefinition, there are in general elements that are direct or recursive targets of two or more elements that are updated by definitions in K . Some implementations might redundantly evaluate such elements twice. In theory, this should not occur since $U(K)$ is a set. However, if $U(K)$ is in practice implemented as a sequence, without collapsing of multiple redundant entries, the implementation could perform more updates than are theoretically necessary. Cartwright [Car99, p.124] states that EDEN is such an implementation:

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 ($\|K\| = 1$) 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.

We shall see in §4.3 that this assertion is not correct, for two reasons. Firstly, EDEN has a complicated evaluation scheduler which makes use of a mix of evaluation/storage strategy 2 (evaluate-on-redefinition) and strategy 3 (evaluate-at-use-when-necessary), the strategy in use being dependent upon current machine operating context. This is described in Chapter 4. This reduces the number of unnecessary evaluations — although this is difficult to observe through black-box testing. Secondly, Eden (the language) contains an `autocalc` facility which can be used to demarcate blocks of redefinitions. When this is used, EDEN (the implementation) effectively avoids redundant evaluation.

As a preliminary to the description of the DAM machine, Cartwright [Car99, §4.3] presents an algorithm “for efficient update of the DMM”, named “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 and all the redefinitions in the block of redefinitions.

The block redefinition algorithm (in this thesis abbreviated to BRA) incorporates Knuth’s algorithm for topological sorting of the elements of a partially ordered set [Knu73, p.262]. Cartwright’s BRA has three phases:

1. Changing D to D' and F to F' . The move from D to D' corresponds to making any necessary changes to the arguments associated with each location. Moving from F to F' corresponds to making any necessary changes to the function associated with each location. These two changes thus correspond to modification of dependencies in the script. As literal values are translated to value functions in LLDN, moving from F to F' corresponds to modification of literal values in the script.
2. Knuth’s topological sort, which achieves two results:
 - (a) Calculation of an optimal evaluation ordering in order to update V to V' . This calculation finds all targets of K in D' as a sequence sorted by level assignment⁶ with no repeated elements.
 - (b) Detection of cyclic dependencies in D — an erroneous condition.
3. Subject to validity, performing the updates, following the ordering determined in step 2, updating V to V' .

Note that even if a redefinition in K is redundant in the sense that it does not change the current definition of a variable, then the relevant updates will still be performed.

If no cyclic dependencies are detected in phase 2, then the state transition from \mathcal{M}_S to $\mathcal{M}_{S'}$ will be successful. If cyclic dependencies are detected in phase 2, the changes made in phase 1 are reverted and the state remains at \mathcal{M}_S , as described earlier for the `protected_update` operation.

⁶This terminology is from [HNC65, p.267] — more explanation in Appendix §3.A, p.178.

The BRA requires some state additional to the representation of the dependency graph to be stored. Two kinds of additional state are required. The first requirement derives from Knuth’s use of counters in the topological sort algorithm to represent the current state of the algorithm’s traversal of the graph. Cartwright incorporates these counters into his BRA, naming them Knuth Counters (KC). One Knuth Counter per definition is required. The second requirement arises from an assumption [Car99, p.126] that the calculation of the set of targets⁷ for any particular reference is “trivial” (i.e. computationally of negligible cost). The efficiency of the BRA depends heavily upon this calculation. It is possible to speed the calculation by effectively precomputing the result, storing doubly-linked source and trigger pointers rather than singly linked source pointers. This is the approach taken in the DAM machine implementation [Car99, p.151].

In connection with Cartwright’s BRA, it may be noted that — because of the characteristics of Knuth’s topological sort — the order of update execution is dependent on the order of redefinitions in the input block K . This has implications where the potential use of the BRA as the basis for a hybrid definitive-procedural machine is concerned and in particular in respect of the extension of the DAM machine to deal with actions (*cf.* the breadth-first sort used in EDEN, described in §4.3.3).

3.1.3 Efficiency and generality of the DMM

The number of element updates (minor transitions) performed during an update operation (major transition) in the DMM block redefinition algorithm, then, “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” [Car99, p.125]. Cartwright [Car99, §4.4] presents two case studies “in which well-known sequential algorithms are represented in a dependency structure”. The first case study consists of a dependency structure that finds the minimum and maximum values in a set of data of size m . Cartwright tabulates the number of element updates required for different $\|K\|$, which range from $2 \log_2 m + 1$ (for $\|K\| = 1$) to $3m - 2$ (for $\|K\| = m$) respectively. The second case study consists of a dependency structure that sorts a

⁷Direct dependents, or ‘d.dependents’, in Cartwright’s terms.

set of data, in a version of the Batcher bitonic merge sort algorithm [GS93], and a similar analysis is given.

But although the BRA is proposed on grounds of efficiency (*cf.* the quote from [Car99] concerning EDEN in the previous section), it requires many calculations to be made to topologically sort the updates and also much additional state to be stored. This is justified (although not explicitly) by a major assumption relating to Cartwright’s application — [Car99, p.122] states: “The arguments presented are based on the premise that each update is expensive in comparison with the calculation of the update ordering”. This may be true if the application involves rendering of complex geometry (the subject of [Car99]), but may not be true in general (for example, for many of the `tkeden` models in the `empub` archive [WRB]).

Another, broader, assumption made by Cartwright and other authors of similar definition maintainers⁸ is implicit. The DMM places no restrictions on evaluation, other than it must not occur whilst an update is in progress. The framework therefore requires the updates to be fully eager: i.e. that the value of everything must be up to date after the update operation has terminated. The DMM therefore uses evaluation/storage strategy 2, evaluate-at-every-redefinition, storing formulae and values. Again, this choice may be appropriate for some applications (e.g. if it is not possible to predict what state will be observed), but strategy 1 or 3 may sometimes be more appropriate.

Two factors determine the efficiency of Cartwright’s BRA: the time taken to perform topological sorting in phases 1 and 2, and the time required for actual evaluation in phase 3. The optimal algorithm for topological sorting is not known. Knuth [Knu73, p.265] however states that the execution time of his topological sort algorithm, on which the BRA is based:

... has the approximate form $c_1m + c_2n$, where m is the number of input relations [pointers from elements to targets], n is the number of objects [elements], and c_1 and c_2 are constants. It is hard to imagine a faster algorithm for this problem!

The aim of the BRA is to effect a block of redefinitions in one single, efficient state transition. In this aim, it is similar to the ADM’s aim of processing multiple redefinitions in a single major transition, conceptually in parallel. However, the `am`

⁸Dominic Gehring’s MoDD [Geh] makes the same assumption.

implementation of the ADM avoids the problem of scheduling the updates by using evaluation/storage strategy 1: evaluate at every use. `am` does not require change to be propagated in an efficient ordering, as the values of definitions are evaluated on demand.

Both the ADM and the DMM use the notion of a machine cycle, implying global synchronisation through some kind of clock: the ADM is explicitly based on machine cycles and the DMM is synchronised through calls to the `update` routine.

3.2 The DAM machine, from the bottom up

The DAM machine is an implementation of the DMM and the BRA, described in [Car99, §5]. Cartwright’s descriptions of the implementation take a top-down perspective, treating LLDN and the DMM as something to be implemented, the implementation being guided by the BRA. The specifics of the implementation do not receive much treatment.

The following section takes the opposite approach, starting at the hardware level and working upwards towards the abstract ideals. This bottom-up approach leads to new insights about LLDN and the DMM, some of which will be explained here. The insights also motivate §5.3, which presents a new DMM-like model.

The material that follows is based largely on practical experience I have had with the DAM machine implementation, rather than [Car99, §5] (which gives little implementation detail) — I have had privileged access to the same machine used by Cartwright.

The bottom-up description in the remainder of this section has the following form. The hardware platform supplied by the specific machine used for implementation is first described. Subsequent subsections discuss the representation of values in the DAM machine, how operators are called by the BRA, what internal data structures the DAM machine requires beyond that described in the DMM and finally how some agent external to the DAM machine can read and change definitive values. The following section then describes !Donald, an application that uses the DAM machine as a basis.

3.2.1 The platform basis for the DAM machine

The DAM machine is an implementation of the DMM and the BRA on the Acorn Risc PC platform. The particular platform that the DAM machine resides on is a Risc PC 700, made in circa 1995. A small amount of technical detail about this particular computing system relevant to the DAM machine implementation is warranted here. Most of the material is derived from [aco92]. Although this detail is not initially required for a good understanding of the DAM machine, it becomes relevant later in this chapter in §3.4 and §3.5.

The Risc PC⁹ 700 contains a single ARM710 CPU, clocked at 40MHz. The ARM processor uses 32 bit words. A word can hold a complete processor instruction, an address reference or a some other data value. The ARM processor has sixteen 32 bit registers accessible when in user mode¹⁰, named R0 to R15. R15 is the program counter (PC). R14 is used as a subroutine link register by instructions such as BL (branch and link). R13 has a special interpretation which is not important here. R0–R12 are free for use as general purpose registers.

The processor has a 32 bit data bus, so an instruction or 4 bytes of data can be fetched in a single step. It has a 26 bit address bus, so 64 Mbytes of memory can be directly addressed. The particular machine in question has 18 Mbytes of physical memory. The logical 64 Mbyte address space is mapped to the physical memory by the custom MEMC (“Memory Controller”) chip.

The computer has video output circuitry which can drive a standard CRT display. Internally, the custom VIDC (“Video Controller”) chip reads data from the video buffer (which is located within conventional addressable RAM) using DMA under control of the MEMC chip. The data is serialised by the VIDC chip into pixels (various modes with varying bits per pixel are available) and then passed through a colour lookup palette before being provided to three digital to analogue converters (DACs) which drive the outputs for the CRT display.

The final custom chip, IOC (Input/Output Controller) manages interrupts and

⁹Expansion boards could be purchased to add another ARM or Intel x86 processor to run in parallel with the standard processor, hence the “PC” ingredient of the model name.

¹⁰There are also 11 other registers accessible in other processor modes such as SVC “supervisor” mode, but these are not relevant here as the DAM machine runs entirely in user mode.

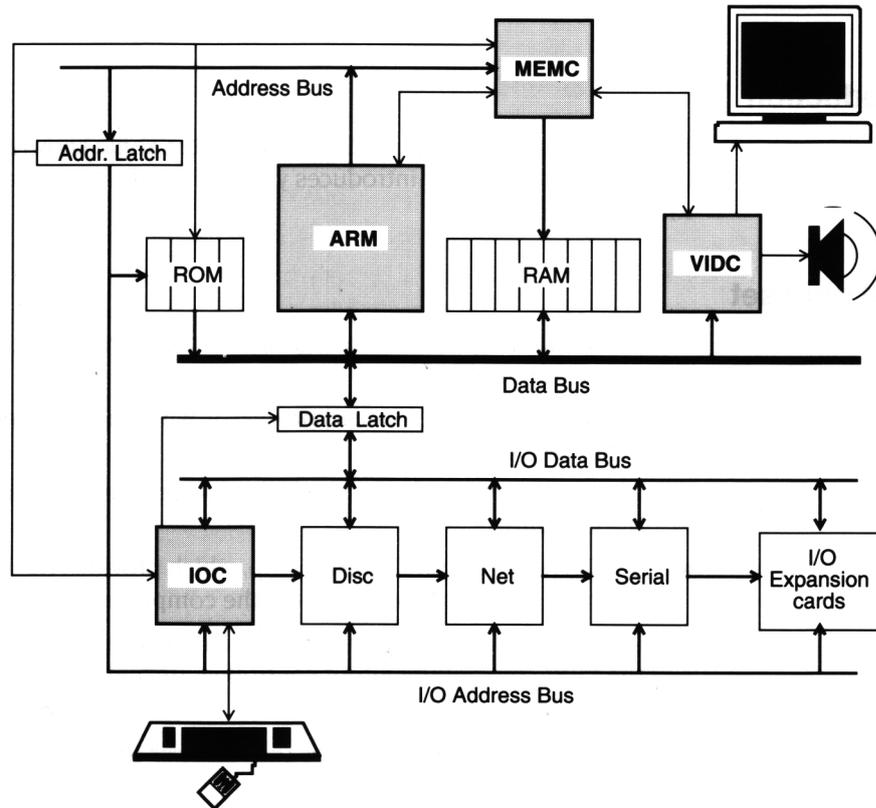


Figure 3.1: Architecture of an Acorn Archimedes 400 series computer, predecessor of the Risc PC (reproduced from [aco92, p.1-10])

peripherals. Figure 3.1 (reproduced from [aco92, p.1-10]) gives a schematic of the architecture.

The “RISC OS” operating system (on the machine in question, version 3.60) is modular and mainly located in ROM. The OS implements clipped graphics drawing routines, a memory heap manager and a cooperatively multitasking GUI among many other features. The main way of accessing OS routines from program code is through the use of an ARM SoftWare Interrupt (SWI) instruction, which changes the processor to SVC “supervisor” mode and branches code execution through a vector into the OS.

The computer system can be programmed directly in ARM assembler, or by using the provided BBC BASIC V interpreter.

3.2.2 Values

In the DAM machine, each definition value is stored in a single word of memory, and so, if considered as an unsigned integer, can have range 0 to $2^{32} - 1$. The values maintained by the DAM machine are stored in a contiguous area of memory of a fixed size known as the *definitive store*. We shall return to these limitations on the geometry of the definitive store in §3.5.

In the abstract DMM, all values are integers and they are assumed to have unbounded range (i.e. the representation problem is ignored at this level of abstraction). In the concrete DAM machine, values are essentially treated as a fixed size opaque type — i.e. each value is a sequence of 32 bits, the particular states of which hold no intrinsic significance to the DAM machine algorithm itself. Knowledge about the specifics of the type representation in use are encoded into the operators.

3.2.3 Operators

The programming user¹¹ of DAM must construct *operators* in ARM assembler, corresponding to *functions*¹² in \mathcal{F} in the DMM. Operators are invoked in phase 3 of the BRA. When an operator is invoked, it is passed a sequence of 10 argument values¹³ and when it terminates, it must return a single value. The problem of determining the necessary 32 bit representations for a particular domain is left to the programming user of the DAM machine who determines the representations as implicitly encoded in the way in which operators treat the values. This separation of concerns is similar to the way that assembly languages deal only with operations on basic types, such as arithmetic on integer and floating point representations and basic array indirection accesses, leaving higher level types to be translated into these atoms by the compiler.

Each register in the ARM processor is 32 bits wide and so can hold a DAM machine value or a machine address reference (recall that the Risc PC's address bus

¹¹Meaning someone who writes a program incorporating the DAM machine, not the implementer of the DAM machine itself or the user of a program which incorporates the DAM machine.

¹²Note the change in terminology from functions to operators, which is intended to emphasise the operational aspect of the DAM machine: operators are *invoked* to bring state back into consistency with the relationship described by the function and arguments.

¹³Actually, references to values: see the next paragraph.

```

.add                ; On entry: R0 = address of value b
                   ;           R1 = address of value c
                   ; On exit:  R0 = b + c
LDR R2, [R0]       ; Load R2 with value at address R0 (b)
LDR R3, [R1]       ; Load R3 with value at address R1 (c)
ADD R0, R2, R3     ; Set R0 equal to R2 + R3 (b + c)
MOV PC, R14        ; Exit with a jump back to DAM

```

Listing 3.1: ARM code for a DAM machine add operator (from [Car99, p.153])

is 26 bits wide). The actual sequence of operations that constitutes a single element update in phase 3 of the BRA is shown below.

1. The BRA loads R0–R9 with the sequence of pointers to argument values within the definitive store region of memory. If there are less than 10 arguments, then the sequence is terminated by a pointer to address zero;
2. The BRA loads R14 with a pointer to the return point (the position in the BRA code where execution should continue after the operator has terminated);
3. The BRA jumps to the start of the operator code (as recorded when redefinitions are introduced to the data structure);
4. The operator code performs its task, using the values pointed to by R0–R9;
5. The operator code loads the single valued result into R0;
6. The operator code jumps to the address in R14, returning control to the BRA.

For example, the ARM code for a binary add operator, taken from [Car99, p.153] is shown in Listing 3.1.

3.2.4 Data structure

The DAM machine operates on a data structure, some of which is a direct implementation of that already described in the DMM and some of which is an extension.

Figure 3.2 illustrates the representation¹⁴. The script used in this example contains a variety of operators. The “top-most” definition, *i*, forms the power of *h* and *a*, so this example will be known as the “power” script.

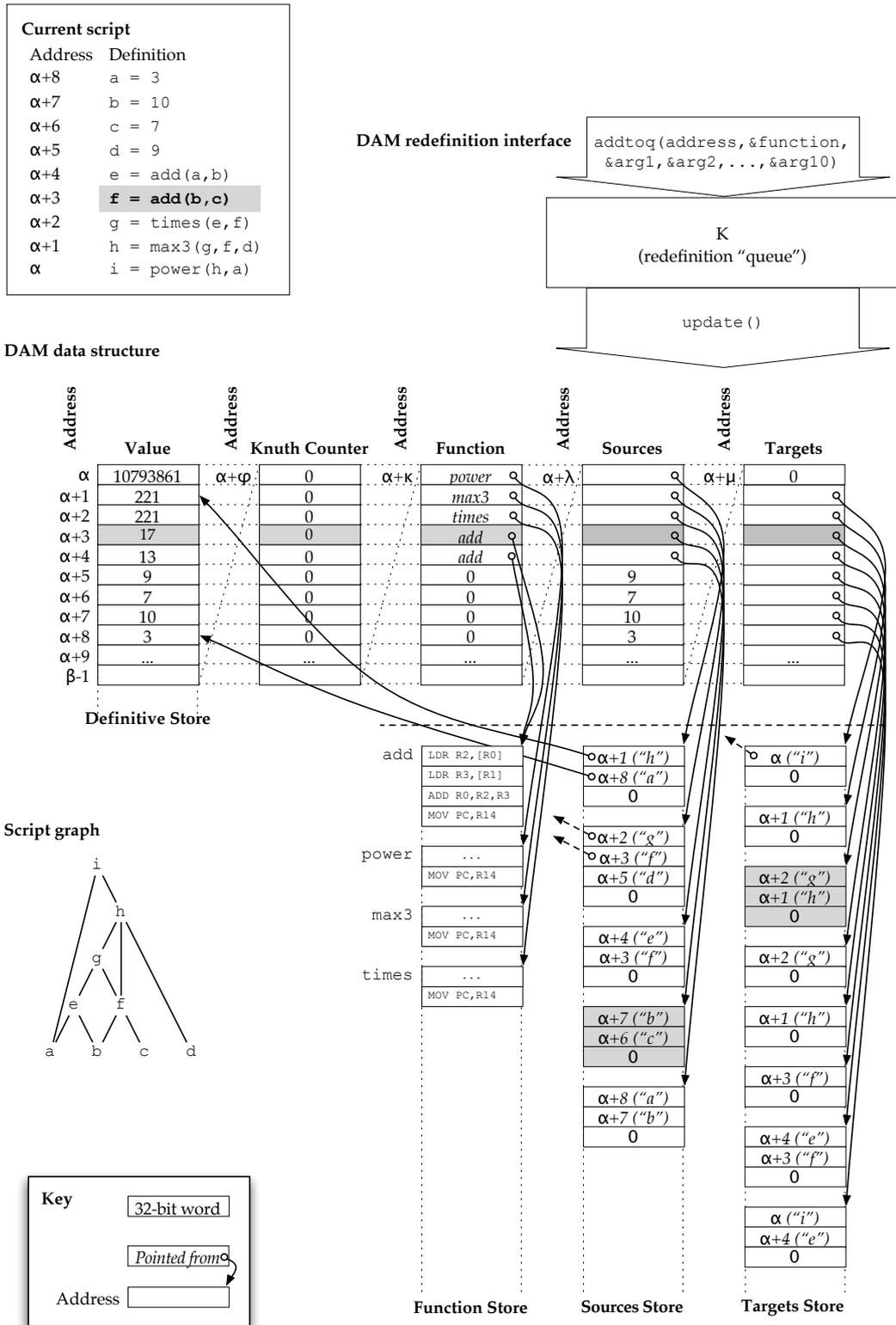
As described earlier in §3.1.1, the DMM maps an identity to a function, a sequence of arguments and a current value. These mappings are accordingly represented in the DAM machine.

- Values are stored in a particular region of memory, ranging α to $\beta - 1$, known as the Definitive Store. Each Value is a 32 bit word.
- A Value stored at address $\alpha + i$ is associated with a Function Pointer, stored at $\alpha + \kappa + i$. Each Function Pointer is a 32 bit word intended to be interpreted as an address, which references the start of the sequence of operator code which is used in the calculation of the Value. Each sequence of operator code is terminated by the 32 bit opcode corresponding to `MOV PC, R14`, as shown for the `add` operator in Figure 3.2 and in the ARM code given earlier for `add`. The memory that holds the operator object code is known as the Function Store.
- A Value stored at address $\alpha + i$ is associated with a Sources List Pointer, stored at $\alpha + \lambda + i$. Each Sources List Pointer is a 32 bit word address which references the start of a sequence of Source Pointers (each being a pointer to a Value). Each sequence is terminated by a pointer to address zero (which is unused by the DAM machine). The memory that holds the sequences of Source Pointers is known as the Sources Store.

In addition to the mappings described in the DMM, to enable use of the BRA, identities are also mapped to Knuth Counter values and to improve efficiency, identities are also mapped to sequences of Target Pointers (as was briefly mentioned on p.107):

- A Value stored at address $\alpha + i$ is associated with a Knuth Counter value, stored at $\alpha + \varphi + i$.

¹⁴The figure is based on [Car99, p.149], but this version illustrates the redefinition interface, eliminates reference by identity within the data structure and uses terminology consistent with this thesis.



- A Value stored at address $\alpha + i$ is associated with a Targets List Pointer, stored at $\alpha + \mu + i$. Each Targets List Pointer is a 32 bit word address which references the start of a sequence of Target Pointers (each being a pointer to a Value). Each sequence is terminated by a zero pointer. The memory that holds the sequences of Target Pointers is known as the Targets Store.

For example, in Figure 3.2, the highlighted definition **f** has a value that is stored at address $\alpha + 3$. We should emphasise here that the DAM machine has no symbol table¹⁵ and does not use symbolic references such as **f**: any occurrences of symbolic rather than numeric references in the figure are markings to aid human interpretation of the diagram. The identity-address mapping (shown alongside the symbolic script in the top left corner) has been deliberately reversed to emphasise this fact. The value is calculated by applying the operator **add** to the sources stored at $\alpha + 7$ and $\alpha + 6$ (**b** and **c**), as represented by the highlighted sequence of Source Pointers. The value is used by the definitions whose values are stored at $\alpha + 2$ and $\alpha + 1$ (**g** and **h**), as represented by the highlighted sequence of Target Pointers.

The data structure is spread across four areas of memory that are allocated in different ways.

- The values are stored in the statically allocated Definitive Store. The store is a fixed size.
- The Knuth Counter values and the Function, Sources and Targets Pointers are stored in statically allocated areas of memory at fixed locations offset from their corresponding Values by φ , κ , λ and μ respectively.
- The operator object code is stored in the Function Store. The location of the Function Store is not specified by the DAM machine, but in the current implementation is statically allocated and the operators are fixed.
- The Sources and Targets Stores are dynamically allocated, using the operating system's heap manager routines, as their size varies according to the number of references made by definitions in the current script.

¹⁵Although !Donald (see §3.3) does.

Each definition therefore requires five words of statically allocated storage (for the Value, Knuth Counter, Function Pointer, Sources List Pointer and Targets List Pointer) as a minimum, plus storage required for the sequences of Source and Target Pointers and the operator object code.

As described so far, the DAM machine can represent definitions that have the values of other definitions as source arguments, but there is no way of setting an explicit value. Literal values (for example, the definition `d = 9` in Figure 3.2) require a special case for representation in the DAM machine. The special representation corresponds to step 4 in the four stage process of translation to LLDN described earlier on p.101. Definitions whose values are defined literally use a special operator `valueX` [Car99, p.151], represented by a Function Pointer to address zero. The literal value is then stored in the Sources List Pointer. This is possible as literal values are source/leaf nodes (see Appendix §3.A, p.178) in the script graph and hence have no need for a list of argument references in the Sources Store. The literal value is copied across to the Value word when necessary by the BRA.

The other special case in the representation is used to describe sink/root definitions (again, see Appendix §3.A, p.178) which are not referenced by any other definitions (for example, the definition `i` in Figure 3.2). In this case, there is no need for a list of Target Pointers in the Targets Store and so the Targets List Pointer holds the special address zero.

3.2.5 DAM execution

The previous section described the static DAM machine state. This section considers dynamic DAM machine transitions.

Figure 3.3 summarises how a transition is made from a state S to the next state S' in the DAM machine. The example is the same “power” example from Figure 3.2. Redefinitions to state S are added to the redefinition queue K (see section §3.1.2) with the `addtoq` routine. When `update` is called, the BRA operates in three phases. In phase 1, the initial changes from K are made to the Operator, Sources and Targets Stores, changing definitions but not recomputing values, moving the state from the initial state S to the “most dirty” state, in a sense to be explained below. In phase 2, an evaluation ordering is calculated for the new script graph, and a check is made

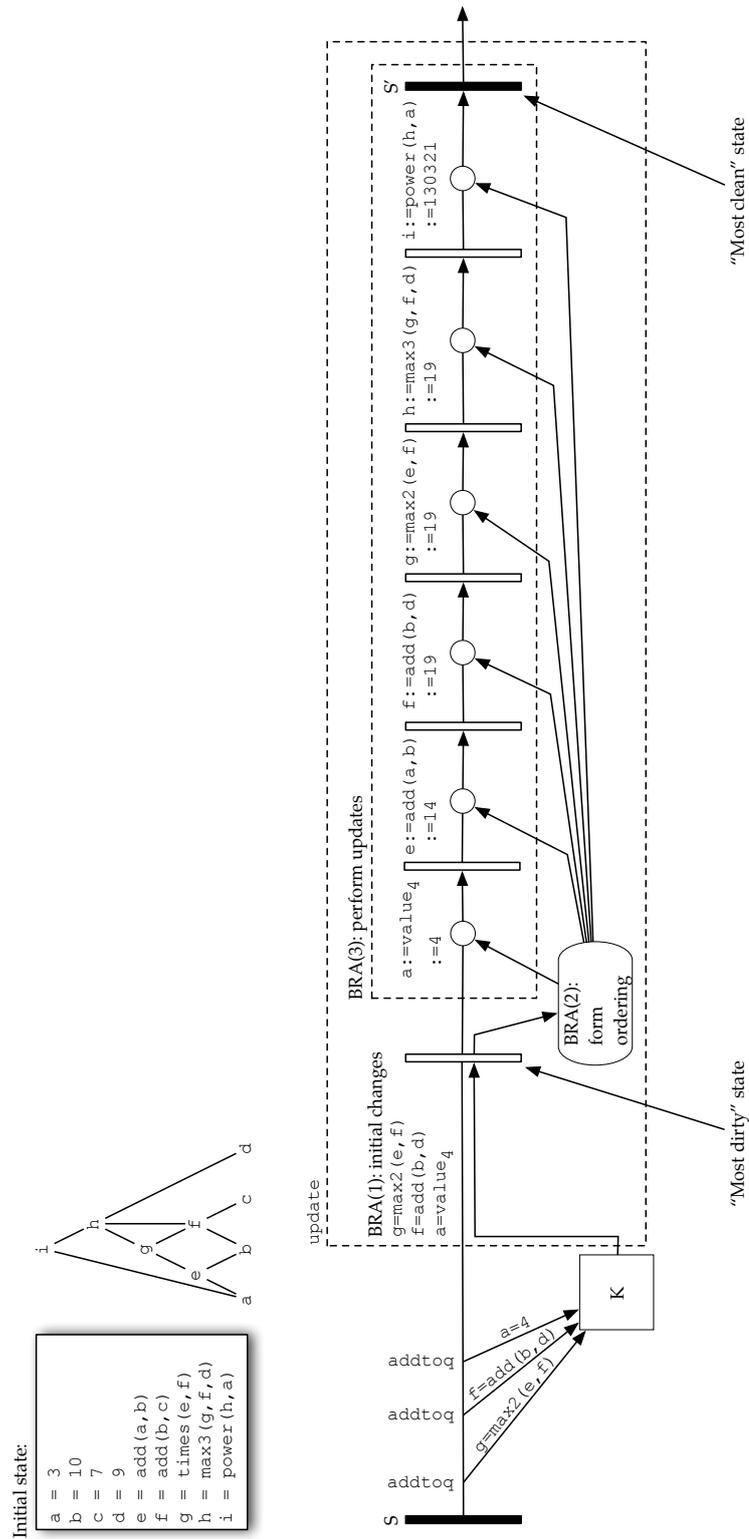


Figure 3.3: State transitions in the DAM machine

for cyclic dependency. In phase 3, the operators are executed sequentially in the ordering calculated by the topological sort in phase 2, finally leaving the values in the “most clean” final state S' .

The number of changes to state made in phase 1 of the BRA is equal to the number of items in K . $\|K\|$ is equal to the number of calls to `addtoq`. (An attempt to redefine the same location more than once results in an error.) The number of updates in phase 3 is $\|U(K)\|$, the value of which can range from $\|K\|$ to n , depending upon the contents of K and the current script. (See §3.1.2 for more explanation.)

The example shows a set of three redefinitions in K :

```
g=max2(e,f)
f=add(b,d)
a=4
```

The BRA calculates the update ordering a, e, f, g, h, i . Note that in this example, there is scope for parallel evaluation: the ordering could be $((a, e) // f), g, h, i$ (that is to say, the update to f could be performed concurrently with the updates to a and e). The DAM machine, however, is implemented on a sequential processor and does not exploit available parallelism.

In the ADM, values are always read from the previous state S during evaluation. S states result from major transitions, and so the values read are taken from a stable state, rather than an unstable S^* state. (See section §2.3.3 for more details.)

In the DMM, the situation is curiously different. We can consider the BRA in terms of a machine that moves from a “dirty” state to a “clean” state. The first phase in the BRA changes F to F' and D to D' . This causes inconsistency between the state as described by the definitions (in F and D) and the state as held by the values. The inconsistency is not limited to just the changed nodes: all nodes in a path above changed nodes are also inconsistent with their definition. (In the example, after the changes have been made to the definitions of g, f and a , then the values of all of a, e, f, g, h and i are inconsistent with their definition.) This state (effectively $S + K$) can be described as the “most dirty”, since a large amount of inconsistency between definitions and values is present. The next two phases of the BRA schedule and then invoke the operators, which (through the BRA) read their sources as input and write their target as output. When phase three of the BRA

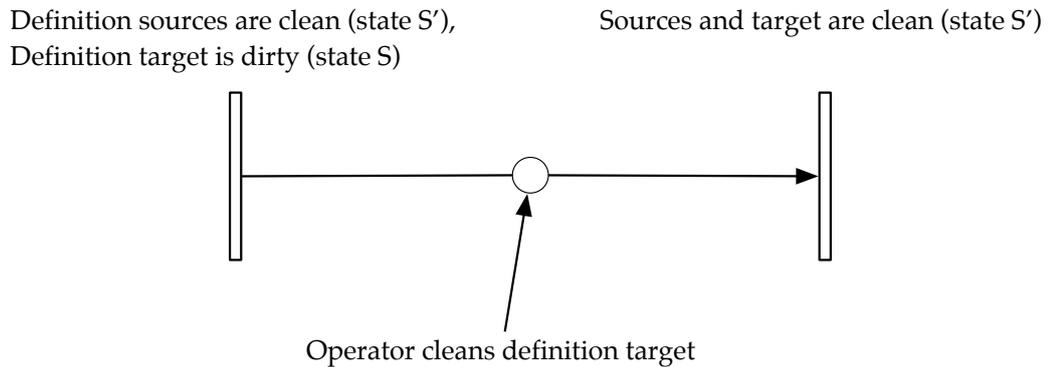


Figure 3.4: A DAM machine operator cleaning partial state

is complete, the state is the “most clean” state S' , where all values are consistent with definition.

The topological sort in the BRA schedules operator invocation so that when an operator is invoked, its source values are in the “clean” state S' and its target value is in the “dirty” state S . The operator then “cleans” its target, moving it into the “clean” state S' . Figure 3.4 illustrates this concept. (Note that the figure is a slight simplification: operators in the DAM machine do not directly affect the state in memory as they interact with the BRA through processor registers, which is itself responsible for updating memory.)

3.2.6 External agency

The DAM machine is intended to be incorporated into other applications, as it is not useful on its own. This section examines the question of how other code can interact with the DAM machine.

The DAM machine redefinition interface is shown at the top of Figure 3.2. The DAM machine maintains a redefinition queue in another area of memory. Code external to the DAM machine can add redefinitions to the queue by calling the DAM machine ‘`addtoq`’ interface routine. The routine takes several arguments, each being passed in an ARM register. Figure 3.2 uses C-style syntax, with the ampersand character `&` denoting a pointer (although the actual code is more likely to be directly coded in ARM assembler). The following arguments to the `addtoq` routine are summarised from [Car99, p.153]:

- R0: address of the Value to be redefined (in range $[\alpha, \beta - 1]$);
- R1: new Function Pointer;
- R2: new Source Pointer 1 (first argument);
- R3: new Source Pointer 2 (second argument);
- R4-R11: Source Pointers 3–10. If there are less than 10 arguments, then the sequence is terminated by a register holding address zero.

In the case of a redefinition to a literal value, the registers must be given the following values before `addtoq` is called:

- R0: address of the Value to be redefined;
- R1: zero (i.e. a `valueX` operator);
- R2: literal 32 bit value.

Calling the `addtoq` routine has no immediate effect on the DAM machine data structure — the routine adds the specified redefinition to the queue. The “queue” is the equivalent of the set K in the DMM (see §3.1.2). The word “queue” is shown here (and in Figure 3.2) in quotes due to this inconsistency between the DMM and the DAM machine in this regard. In the abstract DMM, K is a set. In the DAM machine implementation, K is a queue — notice that the interface routine is named `addtoq`. It is important to keep in mind however that the BRA implements a transition as a set of redefinitions, not a sequence. This is similar to the discussion of command lists in the ADM, in §2.3.3.

The DAM machine `update` routine is the implementation of the BRA. It causes the block of redefinitions — built up incrementally in K through a sequence of calls to `addtoq` — to take effect on the DAM data structure atomically. This can be compared to the way in which a sequence of multiple database updates can be made to take effect on the database atomically through the use of the SQL `COMMIT` command. If any of the redefinitions in K would introduce cyclic dependency, the update is not performed, as the *protected_update* operation introduced in the DMM specifies. After the call to `update`, the redefinition queue is empty.

The two routines `addtoq` and `update` are the only DAM machine interface routines documented for use by external code. How then is external code to read DAM machine state? Cartwright states [Car99, p.155]:

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.

External code should therefore read directly from the Definitive Store. The unstated assumption is that this only happens whilst an update is not in progress. As long as the external code is not running in an interrupt routine, on the test platform (an unexpanded uniprocessor Risc PC), this is a reasonable assumption. Interrupt routines aside, such a machine can be considered to be a single agent, playing a potential multiplicity of roles, but only one role at any given instant in time. When executing the `update` routine, such a machine is by definition not executing external code that could read from the Definitive Store.

Another way that external code can effectively “read” from DAM machine state is to include code in the operator routines that “writes” to state external to the DAM machine, in a type of operator “side effect”. This technique is used in the !Donald application (see §3.3 below), but has some intrinsic problems as we describe in section §3.3.4.

Can and should external code read and write from other parts of the DAM machine data structure? The above quote implied that external code should not write to the Definitive Store, but when talking about extending the DAM machine for dynamic data structures, [Car99, p.159] continues:

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.

A relevant concern is whether the DAM machine implementation is re-entrant: that is, whether the interface routines `addtoq` and `update` can be called from within an operator.

For most purposes, then, from the point of view of external code, the Definitive Store is considered read-only and the other parts of the DAM machine data structure are private to the DAM machine. Changes are made by using the two redefinition interface routines. In some circumstances, however, it may be necessary to directly read and write to “internal” DAM machine data structures.

3.3 !Donald

!Donald is an application written for the Risc PC architecture that implements the DoNaLD definitive notation for line drawing, using the DAM machine to perform dependency maintenance. It provides an interactive environment in which the DAM machine can be used.

This section is about an important technical point: should side-effect be allowed in definitive operators? Both the DAM machine and the more abstract DMM have a large emphasis on dependency — it is not possible for the user to program agency to the extent that this can be done in the ADM or EDEN. But if the operating system graphics drawing routines are to be used to create displays, some automated agency is required in order to call those routines. !Donald uses “graphical actions”, which are DAM machine operators with side-effect to manifest this agency. This extension of DAM machine operators with side-effect causes various associated problems which are outlined in §3.3.4¹⁶.

This section also relates to high level goals for this thesis. As !Donald includes a definition maintainer, it was hoped that it would enable interaction with meaningful state. This section shows that the interaction it does allow is limited. !Donald has been designed to allow a “top-down” style of use: the type of input is DoNaLD and the DAM machine is hidden from the user. Later sections in this chapter describe extensions to the !Donald application which reveal more of the DAM machine foundation, allowing the application to be used in a novel “bottom-up” manner. Before this can be explained, however, we must first discuss what already exists.

¹⁶Gehring’s MoDD [Geh] definition maintainer Java API, which I used in 1998 [War98] to write an interactive definitive sheep simulator, uses a similar technique, and I encountered similar problems.

3.3.1 !Donald overview

!Donald is the work of James Allderidge, a final year undergraduate working on the project with the assistance of Richard Cartwright. There are only a few sources on !Donald. The primary sources are the author's final year project report [All97] and [Car99, §5.4]. "Enabling Technologies for Empirical Modelling in Graphics" [ABCY98] is a secondary source which mostly summarises the work in [Car99, §5.4]. The writing of this section has been informed by practical experience with the results as well as a reading of the above sources.

The !Donald application embeds the DAM machine into an application which itself is procedurally coded in BASIC and ARM assembler. The application provides several features that the DAM machine itself does not include.

- DAM machine operators which operate on graphical data types (e.g. `distance` finds the Euclidean distance between two Cartesian points);
- "graphical actions" — DAM machine operators which generate graphical output by side-effect;
- the DAMscript assembler-like language¹⁷;
- a symbol table structure which associates variable identifiers with address locations. Each variable identifier is a string of characters. The association mapping is represented as a simple sequentially allocated array (not a more efficient hash table), accessed by some ARM assembler routines which find addresses from names and *vice versa* [All97, p.80];
- a DoNaLD to DAMscript compiler, which compiles DoNaLD script into the lower level DAMscript;
- a DAMscript parser, which interprets DAMscript and calls `addtoq`;
- a front end interface which allows DoNaLD scripts to be loaded and displayed.

¹⁷Cartwright [Car99, p.161] names the intermediate assembler-like language "pseudo-DAM code", the "pseudo" being included presumably to distinguish between the language and the information held in the resulting DAM machine data structure. Allderidge [All97, p.23], the original author of this part of the implementation, names the language DAMscript. This thesis uses DAMscript for brevity.

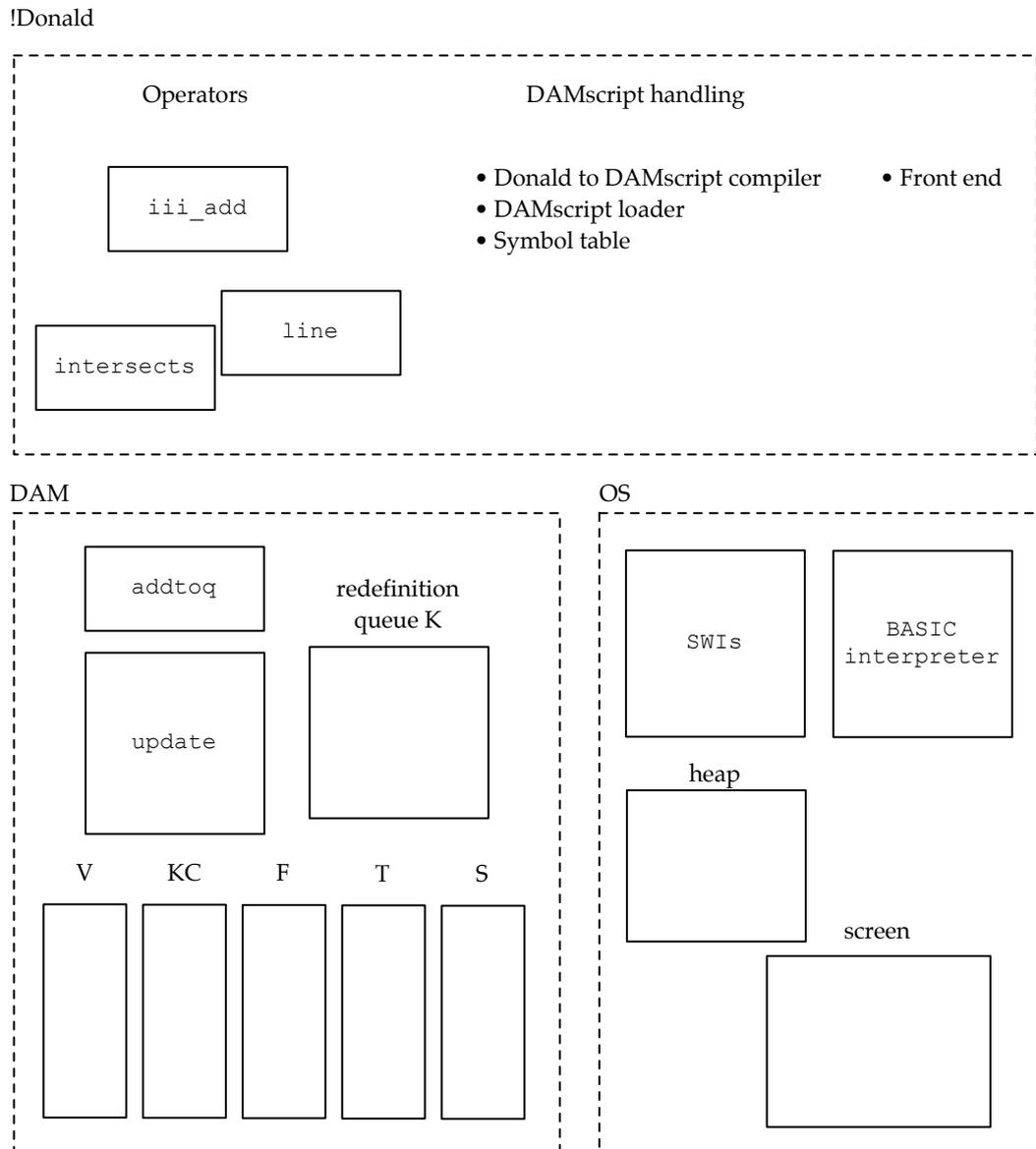


Figure 3.5: Overview of !Donald application components

Figure 3.5 shows an overview of the various components provided by the DAM machine, the operating system and the !Donald application.

The application is shown running in multitasking mode in Figure 3.6. The DoNaLD script file “engine” (1) has been loaded into the !Donald application (2). The graphical result is shown in the display window (3). The contextual menu (4) allows the DAMscript intermediate file to be saved. The “engineDAM” file (5) is

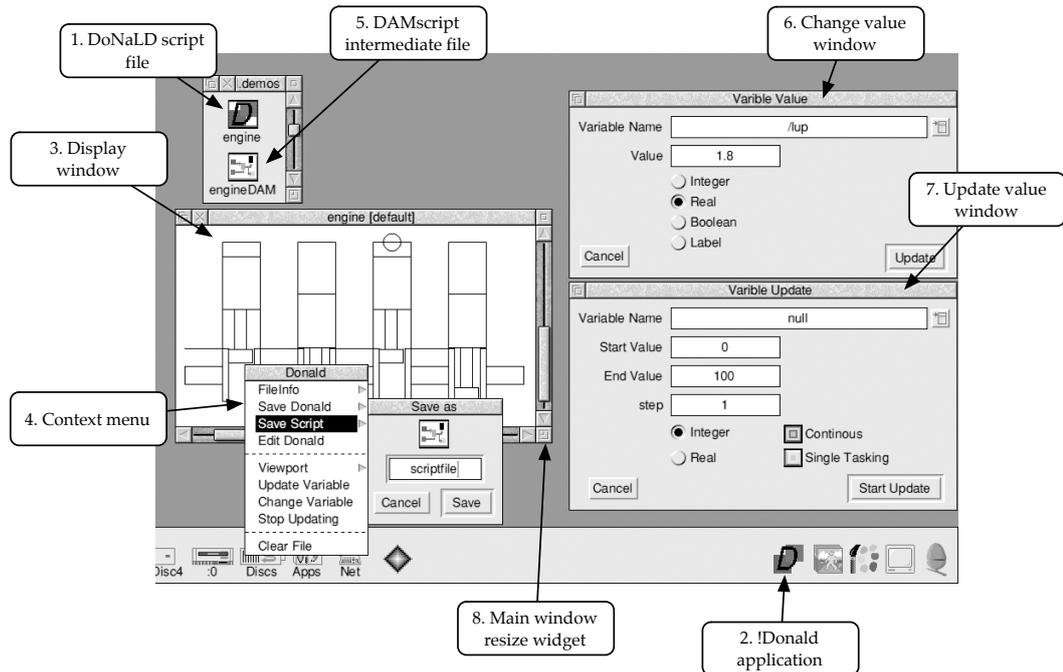


Figure 3.6: The !Donald application

the result of this operation.

The remaining three windows allow different types of redefinition to be made after the initial loading of the DoNaLD file. The “Change value” window (6) (opened from the contextual menu) allows a variable to be redefined to a literal value. In the example, the window has been used to change the `/lup` variable to the value 1.8, which has caused the engine crankshaft to move to the configuration shown in the display window. The “Update value” window (7) allows a variable to be automatically set to a sequence of values, the user providing two limiting values and a step size. This allows animation of DoNaLD graphics to be produced. The animation can be drawn in the display window whilst cooperative multitasking with other applications continues, or the !Donald application can run in “single tasking mode”, taking over the entire screen and temporarily preventing other applications from running.

Considered at a high level of abstraction, the !Donald application passes through two sequential stages of translation (using two separate parsers) in order to render a DoNaLD script on the screen. The first parser translates a DoNaLD script into

an intermediate assembler-like language named DAMscript, in a process which has a close correspondence with compilation of a high level language to assembly code. In the second stage of translation, the assembly-like DAMscript code is parsed and the `addtoq` routine is used to create definitions¹⁸ in the redefinition queue K , in a “loading” process that corresponds roughly to conventional assemble and load operations, generating object code from assembler code and loading this into memory. In the third stage, the `update` routine is called. The definitions are created in the DAM data structure and !Donald’s custom operators are invoked during the BRA, some of which generate output on the screen as a side-effect of their operation. Further calls to `addtoq` and `update` will then cause further state change which might be considered analogous to machine “execution”, although the execution is entirely controlled by the particular definitions that are redefined.

The translation process is illustrated in Figure 3.7. The top of the figure shows a simple DoNaLD script (modified slightly¹⁹ from [Car99, p.162]) that defines two points and a line. The DAMscript output of the compilation pass is shown in the middle of the figure. The bottom of the figure is a screen shot from the DAM machine running in “single tasking” mode, showing a hexadecimal dump of the DAM machine data structure and also the actual graphical line result, drawn between the coordinates {700, 100} and {800, 200} as a result of operator side-effect.

3.3.2 DoNaLD to DAMscript

The DoNaLD notation is defined by Beynon *et al* in [BABH86]. The process of compilation from DoNaLD to DAMscript is described in by Cartwright [Car99, §5.4]. Briefly, a conventional high level language (HLL) compiler reduces sequences of statements in the HLL to sequences of operations specified in assembler. The DoNaLD to DAMscript compiler performs a similar task, but the DoNaLD and DAMscript code are sets of definitions, not sequences of statements. (Some syntactical structures in both languages — for example, `openshape` containers in DoNaLD, or `viewport` statements in DAMscript — require a certain ordering of information,

¹⁸Technically redefinitions, but the DAM data structure is empty at this point.

¹⁹Although the example here is taken from Cartwright, he concentrates mostly on the mapping from DoNaLD to DAMscript. My contribution here is focussed on DAMscript and the levels below, both of which receive little treatment from Cartwright.

obscuring this notion.) The DoNaLD to DAMscript compiler in !Donald is derived from the DoNaLD parser used in `tkeden`, and is written in C, some of which is generated from yacc and lex files.

DAMscript is defined by Allderidge in [All97, p.24]. It provides a number of possible commands, allowing the description in human-readable textual code of:

- `set` commands, where a variable is associated with a literal value;
- DAM machine definitions, where a variable location is associated with an operator and a list of sources;
- `equals` commands, where an alias of a variable name can be constructed;
- `viewport` commands, where the graphical viewport used by drawing operators is set to a particular value until further notice;
- Graphical actions, where an operator with graphical side-effect is associated with a list of sources.

Some BNF for DAMscript is provided in [All97, p.24]. I have edited it for clarity and consistency and included it below.

```
file ::= file command | command
command ::= set | definition | equals | viewport | graphical_action

set ::= set_type variable constant
set_type ::= seti | setf | setb | setc

definition ::= operator_name variable dependent_list
dependent_list ::= dependent_list variable | variable

equals ::= equals variable variable

viewport ::= viewport viewport_name

graphical_action ::= operator_name dependent_list
```

3.3.3 DAMscript to DAM data structure

Generally, a line of DAMscript results in the creation of one definition in the DAM machine data structure. The exceptions to this rule are the `equals` and `viewport` commands, neither of which cause any addition to the DAM data structure.

The DAMscript file is interpreted in two passes by the !Donald application in order to implement forward references: the first pass allocates addresses to symbolic names so that symbolic references encountered during the second pass are all defined.

The *set* and *definition* commands correspond to calls to `addtoq`, although DAMscript takes advantage of the symbol table provided in !Donald, allowing (in fact restricting) variable identities to be specified as symbolic names rather than addresses. The `equals` command does not directly affect the DAM machine data structure (although the notation appears to be introducing additional definitions, this is not the case): it manipulates the !Donald symbol table. The `viewport` command causes modification of some state internal to the assembler/loader stage, which is later used when graphical actions are created.

The DAMscript *graphical_action* command is similar to the DAMscript *definition* command. Definitions are depicted in DAMscript, as shown in the BNF above, using the form *operator_name variable dependent_list*. The definition ‘a is b+c’ would therefore be written in DAMscript as `add a b c`. Graphical actions are shown in DAMscript using the syntax *operator_name dependent_list*. An example action in DAMscript is `line x1 y1 x2 y2`.

There are two main differences between the DAMscript *definition* and *graphical_action* commands. Both commands create a DAM definition, which associates a sequence of input locations with an output location. When a *graphical_action* is created, three extra locations — representing viewport (V), x offset (X) and y offset (Y) respectively — are added to the input sequence. The actual location of V is related to the active viewport at time of creation of the line, and the values at the locations that X and Y reference describe the top left coordinate of the

DoNaLD graphical output window. These extra arguments are added automatically for operators that are known to be *graphical_actions* at the assemble/load stage. The extra arguments²⁰ are therefore not visible in the DAMscript code. The DAMscript command:

```
line x1 y1 x2 y2
```

would be created in the DAM machine data structure as:

```
line x1 y1 x2 y2 V X Y
```

The two types of commands also differ in terms of operator output. Although DAMscript *graphical_action* commands create a DAM machine definition which does have an output value in the Definitive Store, the output value is later ignored by !Donald. In addition, the location of output values from graphical actions are not stored in the !Donald symbol table and so cannot be referenced by other definitions.

Definition and graphical action operators are coded in ARM assembler. The !Donald application is provided with three pieces of information about each operator: the start address, the number of arguments required, and whether it is a graphical action²¹.

3.3.4 Graphical action execution in !Donald

The BRA is designed to invoke operators in the correct sequence in order to produce a “clean” Definitive Store state, as detailed earlier in §3.2.5.

The introduction of graphical actions, however, has complicated the matter of scheduling. A need for redrawing the DoNaLD graphics can originate from two different kinds of stimulus: internally within the !Donald application or externally from the operating system.

²⁰Note that the total number of arguments is limited to 10, including the hidden V, X and Y if appropriate.

²¹This last piece of information being encoded using the boolean variable ‘has_depend’.

Internally to !Donald, the user might make a change to a DoNaLD variable using one of the !Donald Variable Update windows. If that change causes a subset of the geometry to change, then the graphical actions must be invoked to redraw the graphics. However they must be invoked at the correct time. The operating system graphical routines implement a global “clipping plane” setting. Calls to graphical routines will have no effect if they attempt to draw outside the current clipping planes (although if some portion of the result is within the current clipping planes, that portion will appear). If the clipping planes are not set correctly at the time that the graphical actions are invoked, at best the graphical actions will have no effect. At worst, they may draw over some part of the screen that is not currently within a !Donald window, corrupting the screen output.

External events may cause the operating system to request a partial screen redraw from !Donald. An example of an external event here would be if the user used the resize widget (marked (8) in Figure 3.6, p.126) to enlarge the !Donald window, revealing geometry that was not previously on the screen. Another example would be if the user dragged a window across the !Donald window, obscuring and then revealing portions of geometry. The operating system implements screen updates by first splitting the out-dated area of pixels into a set of rectangles, then taking each rectangle one at a time, sets the graphical routine’s clipping planes to the rectangle, then sends a “redraw request” event to the application responsible for that area, continuing thus with the remaining rectangles until the screen is updated. !Donald may therefore receive a “redraw request” event resulting from some external causation.

!Donald must therefore ensure that graphical action operators are invoked only when a redraw request is made by the operating system. If there is an internal motivation for a redraw, !Donald must first ask the operating system to make a redraw request. Only when the operating system responds with the request may !Donald invoke the necessary operators.

To illustrate how the graphical actions are invoked at the correct time, the DoNaLD script shown in Listing 3.2 was created.

```

%donald
point b1
b1 = {700, 100}
label lab1
lab1 = label("label one", b1)
viewport two
label lab2
lab2 = label("label two", b1)

```

Listing 3.2: A simple DoNaLD script with multiple viewports

```

seti intervar2 700
seti intervar3 100
equals intervar0 intervar2
equals intervar1 intervar3
label intervar6 intervar7 intervar8
setc intervar9 "label one"
equals intervar6 intervar0
equals intervar7 intervar1
equals intervar8 intervar9
viewport two
label intervar12 intervar13 intervar14
setc intervar15 "label two"
equals intervar12 intervar0
equals intervar13 intervar1
equals intervar14 intervar15

# this last line added manually to allow the store to be examined
print_store

```

Listing 3.3: DAMscript translation of the DoNaLD script shown in Listing 3.2

The DoNaLD script is translated by !Donald into the DAMscript code shown in Listing 3.3.

In !Donald, DoNaLD scripts have an implicit `viewport default` command at the top. The script therefore creates two textual labels, one in viewport `default`, and the other in viewport `two`. Only one viewport is displayed at a time in !Donald. The currently active viewport is selected using a contextual menu option, as shown in Figure 3.8.

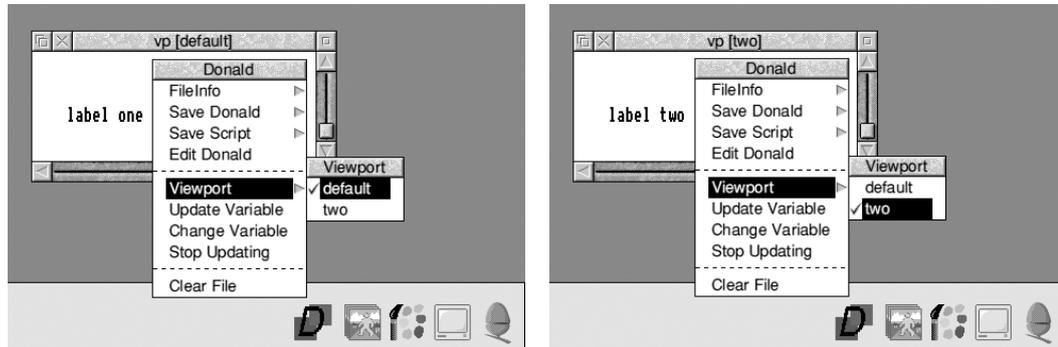


Figure 3.8: Selecting a !Donald viewport

DEFINITIVE STORE - TotalDefs=12					
Address	Value	KC	Op	Targets	Sources
00027D18	00000000	00000000	00000000	000D2BC0	00000000
00027D1C	00000000	00000000	00000000	000D2C10	00000000
00027D20	00000000	00000000	00000000	000D2C18	00000000
00027D24	000002BC	00000000	00000000	000D2BD8	00000000
00027D28	00000064	00000000	00000000	000D2BE0	00000000
00027D2C	0024FFF0	00000000	00000000	000D2BB8	00000000
00027D30	00250024	00000000	00000000	000D2BE8	00000000
00027D34	000002BC	00000000	001C8A78	00000000	00117180
00027D38	00000001	00000000	00000000	000D2C08	00000000
00027D3C	00000001	00000000	001C8A78	00000000	00117180
00027D40	FFFFFFFF	00000000	001C8AF8	00000000	001171E0
00027D44	0000003A	00000000	00000000	00000000	00000000

TARGETS STORE		
Address	Value	Pointer
000D2BA8	00027D34	00000000
000D2BB0	00027D34	00000000
000D2BB8	00027D34	00000000
000D2BC0	00027D34	00000000
000D2BC8	00027D34	00000000
000D2BD0	00027D34	00000000
000D2BD8	00027D3C	000D2BA8
000D2BE0	00027D3C	000D2BB0
000D2BE8	00027D3C	00000000
000D2BF0	00027D3C	00000000
000D2BF8	00027D3C	000D2BC8
000D2C00	00027D3C	000D2BD0
000D2C08	00027D40	000D2BF0
000D2C10	00027D40	000D2BF8
000D2C18	00027D40	000D2C00

SOURCES STORE		
Address	Value	Pointer
00117180	00027D24	00117188
00117188	00027D28	00117190
00117190	00027D2C	00117198
00117198	00027D18	001171A0
001171A0	00027D1C	001171A8
001171A8	00027D20	00000000
001171B0	00027D24	001171B8
001171B8	00027D28	001171C0
001171C0	00027D30	001171C8
001171C8	00027D38	001171D0
001171D0	00027D1C	001171D8
001171D8	00027D20	00000000
001171E0	00027D38	001171E8
001171E8	00027D1C	001171F0
001171F0	00027D20	00000000

label two

Figure 3.9: A screen shot of !Donald with Listing 3.3 loaded, showing the DAM store

Loading the DAMscript code into !Donald²² and viewing the result in single-tasking mode, whilst continuously updating a new variable “null” in unit steps between 0 and 100, gives the screen shot shown in Figure 3.9. The constantly incrementing value for “null” is shown at address 0x27D44²³, and had the value 0x3A (61 in decimal) when the screen shot was taken²⁴. Viewport two is being shown, as evidenced by the visibility of “label two”.

The hexadecimal store dump can be interpreted back into the DAM data structure diagram shown in Figure 3.10. The three graphical actions are shown at the top. In the diagram, attributes of definitions are marked with the characters ‘r’, ‘m’, ‘o’, ‘a’ and ‘v’, representing reference, meaning, operator, address and value respectively²⁵. The actions `lab1` and `lab2` at the top of the figure both take six arguments. From left to right, these are: the x and y positions of the label within the viewport, a pointer to the label text, a viewport variable and the x and y offsets representing the current viewport screen origin coordinates. The first three arguments are stated explicitly in the DAMscript code — the last three “hidden” arguments are added automatically during the !Donald “load” phase. The rightmost `print_store` graphical action (which I added manually to the file shown in Listing 3.3) takes only the three hidden arguments and requires no arguments at the level of DAMscript code.

Below the graphical actions are shown the dependencies, which in this case all happen to be literal values and hence use the “value operator” special case. The leftmost two definitions are the x and y values of the DoNaLD point `b1`. These definitions each have several reference aliases constructed by the DAMscript `equals` command, which are artefacts of the DoNaLD parsing process. Next on the left are two definitions whose values point to the label strings (which are not DAM machine definitions). Rightmost are four “hidden” definitions, which have been added automatically. From left to right, these are: the two viewport variables and the current viewport screen origin coordinates. Finally, the “null” variable

²²Which is possible due to an extension I authored — see §3.4.1.

²³The prefix 0x denotes hexadecimal, following the language C, as does the prefix &, used later, following the language BBC BASIC.

²⁴Also made possible by another extension I authored.

²⁵As noted previously, graphical actions in !Donald are DAM machine dependencies, but the address and the value are not stored in the symbol table and so are not accessible from other DAMscript commands, hence they are shown in parenthesis in the diagram.

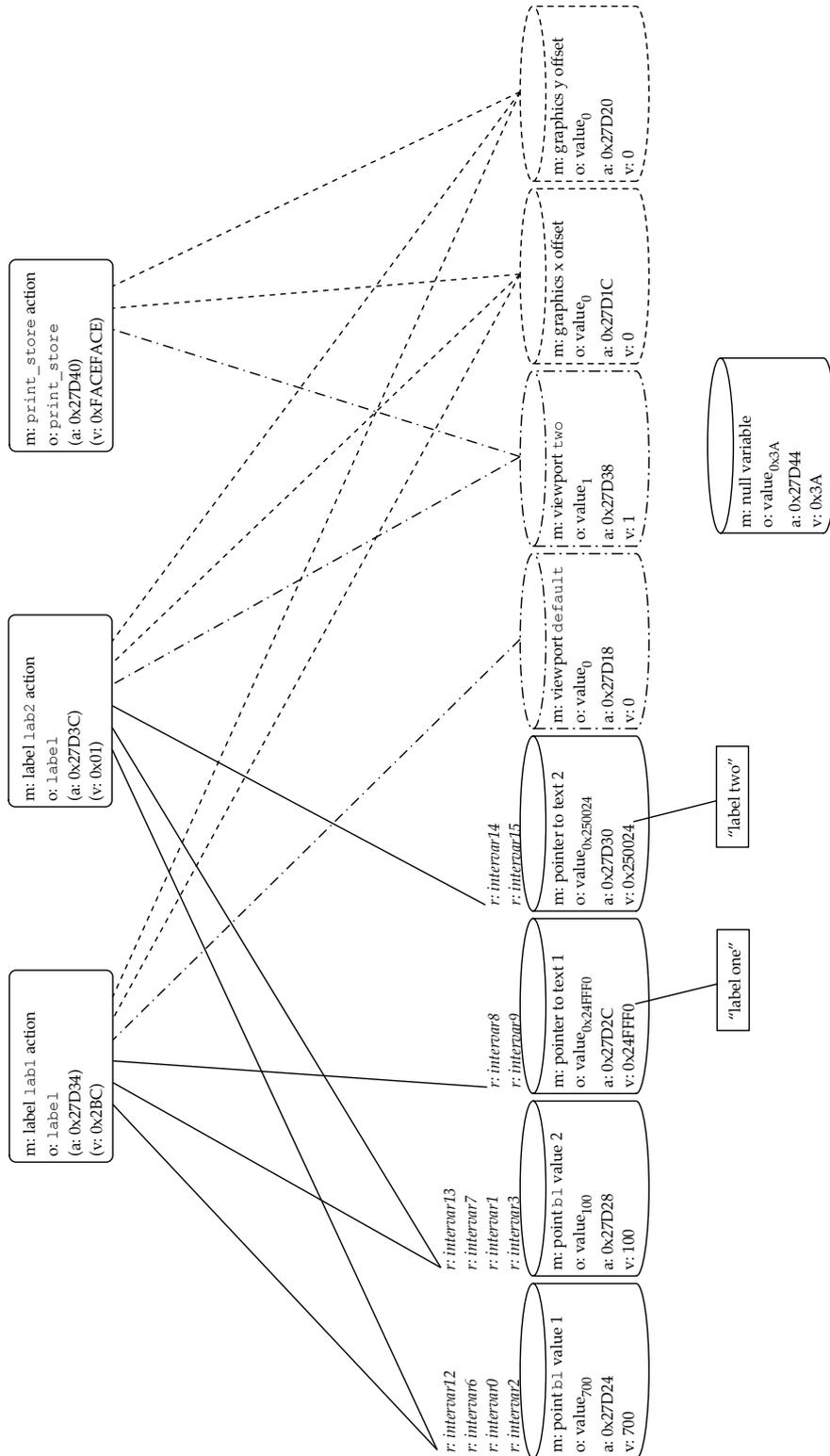


Figure 3.10: DAM machine data structure shown in Figure 3.9

introduced whilst viewing the graphical output in single-tasking mode is shown. It is an “orphan dependency” which is not used by any other dependencies.

The viewport variables are used to control when the graphical action operators are invoked, giving some control over their side-effect. Allderidge [All97, p.101] explains the process of re-displaying the DoNaLD viewport window (which I have rewritten for clarity and consistency):

1. The viewport variable for the active viewport (i.e. a single variable that is a source for all graphical actions in a particular viewport) is set to 1 using `addtoq`. All viewport variables for other viewports are set to 0.
2. The x and y viewport screen origin offsets (which are sources for all graphical actions in all viewports) are set to the correct values using `addtoq`.
3. The `update` routine is called.
4. All viewport and offset variables have been redefined. These variables have all the graphical actions as targets, and so the BRA invokes *all* the graphical action operators. (Recall that the BRA does not optimise in the $D' = D$ and $F' = F$ case.)
5. The first act of each graphical action operator is to check its viewport variable. If the value is 0, the operator exits without drawing. If the value is 1, the operator enacts its side effect, drawing the relevant graphics.

For example, the ARM code for the graphical action `line` operator, taken from [All97, p.137] is shown in Listing 3.4.

In summary, then, !Donald introduces graphical side-effect without fundamentally changing the DAM machine. This introduces three synchronisation problems that must be solved through the scheduling of operator action: 1) a graphical side-effect must not occur before the state it is graphically presenting is “clean” (in state S'); 2) it must be possible to invoke the graphical actions on demand from the operating system, and 3) it must be possible to inhibit graphical actions that exist in the dependency structure but do not correspond to an active viewport.

Allderidge solves these problems by introducing “graphical actions”, which are DAM operators with graphical side-effect and extra “viewport” variables. Each

```
.line          ; On entry: R0 = line start X coordinate
               ;          R1 = line start Y coordinate
               ;          R2 = line end X coordinate
               ;          R3 = line end Y coordinate
               ;          R4 = viewport variable
               ;          R5 = viewport X offset
               ;          R6 = viewport Y offset

CMP R4, #1
MOVNE PC, R14  ; Exit if line not in the active viewport

add R0, R0, R5
add R1, R1, R6
add R2, R2, R5
add R3, R3, R6 ; Add offsets to start and end coordinates

mov R5, R2
mov R2, R1
mov R1, R0
mov R0, #&04
swi OS_Plot    ; Call OS routine: move to start coordinate

mov R0, #&05
mov R1, R5
mov R2, R3
swi OS_Plot    ; Call OS routine: draw to end coordinate

MOV PC, R14    ; Exit with a jump back to DAM
```

Listing 3.4: ARM code for the !Donald line graphical action operator

graphical action operator has as source variables the state it is graphically presenting and one of these viewport variables as a hidden source variable. The three problems are then solved: 1) because operators are invoked by the BRA only when their sources are in state S' ; 2) because changing the viewport variable forces evaluation of the associated graphical actions, and 3) because every graphical action is given responsibility for first checking that their viewport is active.

There are many problems with this solution, however.

- Forced invocation is coarse-grained: Graphical action side effect occurs for *all* actions in the currently active viewport, and no others, when the above invocation process is followed.
- The scheme seems inefficient:
 - Extra variables are introduced, together with many references to them;
 - All graphical operators must first check that their result is within the active viewport.
- The scheme is sensitive to changes in the underlying BRA implementation. The purpose of the BRA implementation is to update the state to S' — it is a side effect that it achieves this by ordering the updates sequentially. Optimisation of the BRA (e.g. omitting updates where $F = F'$ and $D' = D$) in this scheme would then change the observed result.

The graphics implemented by the current !Donald are also rather simple — only 2D line plotting is implemented. More sophisticated graphics would in some cases require more sophisticated scheduling. For example, if DoNaLD object layering were to be specified and implemented (which is an important feature if solid polygons were made available), more finely grained control would be required in order to ensure that the objects are drawn in the correct order, furthest away first.

The scheduling implemented by the BRA is designed to produce “clean” definitive state, not invoke graphical actions. It is difficult to take a machine designed with a focus purely on dependency in this way and extend it to solve problems involving patterns of agency that were not preconceived during the design. A different type

of scheduling, producing a different pattern of agency, is also required when higher-order definitions are introduced — see §3.4.3. EDEN was specifically designed to separate dependency and agency concerns and is investigated in §4.3.

3.4 Definitive programming in !Donald2

3.4.1 Extending !Donald

In the original !Donald application, once a script is loaded, the only interaction possible is through the windows marked (6) and (7) in Figure 3.6 (p.126), which provide the ability to assign literal values to variables and to automatically set a variable to a sequence of values.

In order to experiment with the DAM machine at a lower level, I extended the application, renaming it !Donald2. The extensions include facilities to interact with !Donald2 using DAMscript, opening up possibilities for changing the definitive structure after the DoNaLD script has been loaded, going beyond the limited form of redefinition to literal values provided previously. The extensions include the following:

- The ability to load in DAMscript (rather than just DoNaLD) code;
- The ability to *redefine* a variable (rather than just set a new literal value) in a parameterised dialog box;
- The ability to interactively enter a line of DAMscript code, where any of the DAMscript commands can be used;
- DAMscript comments (on lines which start with #).

Facilities were also added to view the internal state and save this view. These facilities were used to generate the hexadecimal data structure dumps analysed in the previous section.

- A `print_store` graphical action (that exploits code written by Cartwright but which was not previously available in !Donald) to show the internal DAM machine data structure in a rudimentary way;
- The ability to take a screenshot whilst in single tasking mode.

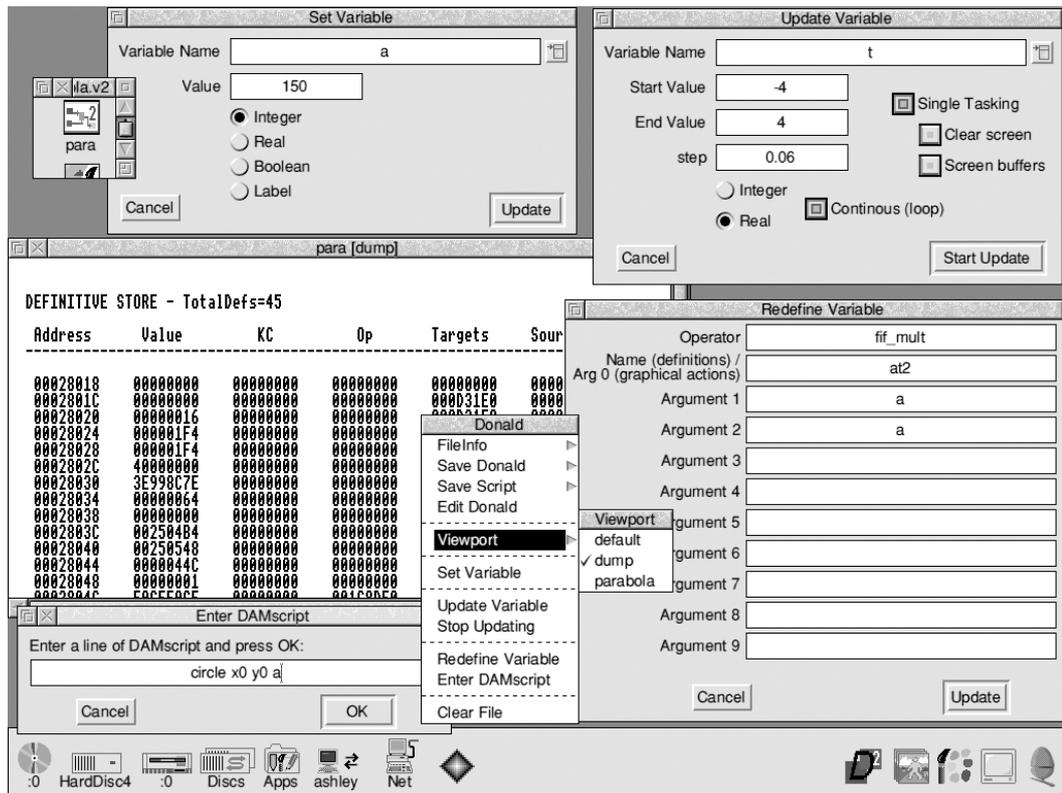


Figure 3.11: The !Donald2 user interface (compare Figure 3.6, p.126)

3.4.2 Using raw DAMscript: the parabola example

The screen shot in Figure 3.11 shows !Donald2, illustrating the new features in use. The figure can be compared with Figure 3.6 (p.126) which shows the older !Donald version.

The extensions allowing the use of DAMscript code permit !Donald2 to be used entirely at the DAMscript level, without using the DoNaLD notation at all. The DAM machine operators available in !Donald2 are described in Tables 3.2, 3.3 and 3.4 (which are derived from some joint work in April 2000 with Carlos Fischer). Some operator names describe their output and input types through the inclusion of the characters ‘b’, ‘i’ and ‘f’ in the name (output type first), meaning boolean, integer and floating point respectively. Where names in the table use the characters ‘X’, ‘Y’ and ‘Z’, meaning any of the above types, this abstraction is merely a device used to shorten the table — the actual operators do not use abstract types.

Boolean	Integer	Floating point	Integer and floating point
bXY_eq	iii_add	fff_add	XYZ_add
bXY_neq	iii_sub	fff_sub	XYZ_sub
bXY_gteq	iii_mult	fff_mult	XYZ_mult
bXY_gt	iii_div	fff_div	XYZ_div
bXY_lt	iii_mod	fff_mod	XYZ_mod
bXY_lteq	i_sqr	f_sqr	
bbb_and	i_ln	f_ln	
bbb_or	i_exp	f_exp	
bb_not	i_sub	f_sine	
		f_cos	
		f_tan	
		f_asine	
		f_acos	
		f_atan	
		f_int	
		iflt	
		f_sub	

Table 3.2: Numerical !Donald2 operators

Boolean geometry	Polar	Geometry
pt_betwn_pts	pi_polar1	midpoint1
includes	pi_polar2	midpoint2
cinci	pf_polar1	intersect1
collinear	pf_polar2	intersect2
intersects		perpend1
separates		perpend2
distlargerpoint		perpend3
distlargerline		perpend4
distsmallerpoint		distance
		rot_pointssx
		rot_pointsy

Table 3.3: Geometrical !Donald2 operators

General	Graphical actions
itos	line
rtos	circle
if	ellipse
fn	label
gfn	print_store

Table 3.4: Miscellaneous !Donald2 operators

In order to demonstrate the way in which !Donald2 can be used at the DAMscript level, the “parabola” DAMscript shown in Listing 3.5 below was constructed using the operators described in the above tables.

```
1 # parabola in DAMscript
2 # Ashley Ward (ashley@dcs.warwick.ac.uk) December 2003
3 # with assistance from WMB and Russell Boyatt
4
5 # See the DAMscript BNF for a full explanation of syntax - but it may help
6 # understanding to note that definitions are 'operator target sources'
7
8 # do a hexadecimal store dump in a "dump" viewport to give a rudimentary
9 # display of the internal state
10 viewport dump
11 print_store
12
13 # do the rest of this script in another viewport
14 viewport parabola
15
16 # origin is (x0,y0), portion of screen used is (0,0) to (2x0,2y0)
17 seti x0 500
18 seti y0 500
19 iif_mult 2x0 x0 two
20 iif_mult 2y0 y0 two
21 setf two 2.0
22
23 # these are the variables that we /expect/ to be changed - although
24 # anything in this script can of course be redefined at any time
25 setf t 0.1
26 seti a 100
27
28 #  $x = x_0 + a \cdot t^2$ 
29 fif_add x x0 at2
30 fff_mult t2 t t
31 fif_mult at2 a t2
32
33 #  $y = y_0 + 2at$ 
34 fif_add y y0 2at
35 fif_mult at a t
36 fff_mult 2at two at
37
38 # define a symbol named '0' that happens to have the value 0
39 seti 0 0
40
41 # the y axis
42 line x0 0 x0 2y0
43 # the x axis
44 line 0 y0 2x0 y0
45 # a vertical line at  $x_0 - a$ 
46 line x0minusa 0 x0minusa 2y0
47 iii_sub x0minusa x0 a
```

```

48
49 # form integer versions of x and y so they can be used as pixel positions
50 i_flt xi x
51 i_flt yi y
52
53 # the point p
54 setc ptext "p"
55 label xi yi ptext
56 # the horizontal line
57 line x0minusa yi xi yi
58 # the radial line
59 line aplusx0 y0 xi yi
60 iii_add aplusx0 a x0
61
62 # if p is not between (0,0) and (2x0,2y0), it is off the screen
63 bii_lt xilt0 xi 0
64 bii_lt yilt0 yi 0
65 bbb_or plt0 xilt0 yilt0
66 bii_gt xigt2x0 xi 2x0
67 bii_gt yigt2y0 yi 2y0
68 bbb_or pgt2xy0 xigt2x0 yigt2y0
69 bbb_or poffscreen plt0 pgt2xy0
70
71 # warning label that appears if p is off the screen
72 setc warntext "p is off screen"
73 label warnx warny warntext
74
75 # could use 'equals warny y0', but that creates a symbol table alias, not
76 # a new definition
77 iii_add warny y0 0
78
79 # "hide" the warning label if necessary by positioning it off the screen
80 seti largex 1100
81 # in C, the following would be 'warnx = poffscreen ? 0 : largex'
82 if warnx poffscreen 0 largex

```

Listing 3.5: The “parabola” DAMscript

The “parabola” script defines lines describing x and y axes (lines 44 and 42), a vertical line a units to the left of the origin (line 46), and two lines (lines 57 and 59) to the point p , where a textual label is located (line 55). The point p takes the coordinates $\{x, y\}$, where $x = x_0 + at^2$ and $y = y_0 + 2at$.

The script demonstrates the use of floating point, integer and boolean values. The above formulae are calculated using floating point values, but the results must be first cast into versions that use the integer type (lines 50 and 51) as graphical actions require pixel coordinates to be of integer type.

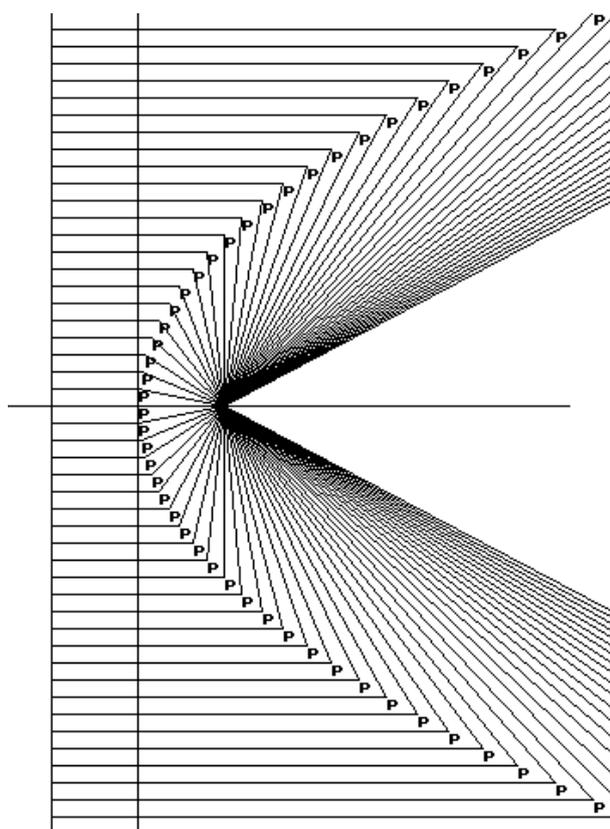


Figure 3.12: A screenshot of the parabola

Changing the value of τ causes the point p to follow the parabola described by the above equations, as shown in screenshot Figure 3.12. The screenshot was taken by varying the value of τ between -4 and $+4$ with a discrete step of 0.06 , using the Update Variable window shown at the top right of Figure 3.11. Additionally, the ability to disable clearing of the screen before each update in single-tasking mode was added to !Donald2, controllable with a tick box in the Update Variable window. This enabled the resulting image for each different value of τ to persist in the display and the parabola to be viewed.

The value τ is not the only item in the script that can be changed: the user can redefine any item (except graphical actions, as these have no symbolic name and cannot therefore be redefined at this level). The interface presents several different ways in which changes can be made.

The Set Variable window allows a variable to be assigned a new literal value. Figure 3.11 shows the window set to redefine the variable a to the integer value

150. The type of a variable can also be changed using the Set Variable window. However if the type of a value is changed, the operators of the targets of that value will probably also need to be redefined to reflect their new input. For example, the `fif_mult` operator used to calculate `at` (line 35 of Listing 3.5) interprets the first input, `a`, to be a 32-bit integer value. If the value `a` was changed to a floating point value, `fif_mult` would still interpret the floating point 32 bit representation as an integer value, causing `at` to have an unexpected value.

The Redefine Variable window can be used to give a new definition to an existing variable, create a new variable with a definition or create a new graphical action. It cannot be used to give a literal value to a variable as literal values require the special case `valuex` form of DAM machine operator. Figure 3.11 shows the window set up to redefine the variable `at2` from `at2 = a * t2` to `at2 = a * a`, causing the definition `x` to become `x = x0 + a3`.

Finally, the Enter DAMscript window subsumes both of the above windows, as it can be used to give any DAMscript command: `set`, (re)definition, `equals`, `viewport` or new graphical action. Figure 3.11 shows the window set to create a new `circle` graphical action centred on the origin with radius `a`. The “Enter DAMscript” dialog can be compared to the `tkeden` Input Window: it allows incremental changes to be made to the state. However !Donald2 presently lacks an easy way to view or save the state in DAMscript form, so the interaction possible is rather limited: the current state must be visualised mentally or on paper by the human user.

Notice that the line ordering of the script has no importance. (The line numbers in Listing 3.5 have been added for these printed pages for ease of reference.) For example, the value of `two` is referenced (line 19) before it is defined (line 21). The `viewport` command, however, breaks this rule. It sets the sequentially interpreting machine into a particular viewport context, and all definitions that follow it are implicitly in that context. Notice from the contextual menu in Figure 3.11 that three viewports have been defined: the `default`, which is empty; `dump`, which holds the `print_store` action defined at the head of the script and `parabola`, which holds the results of the graphical actions defined in most of the script. If it were desired to redesign the DAMscript notation so as to emphasise the unordered set nature of the input text, a more verbose form, stating the context explicitly for each definition

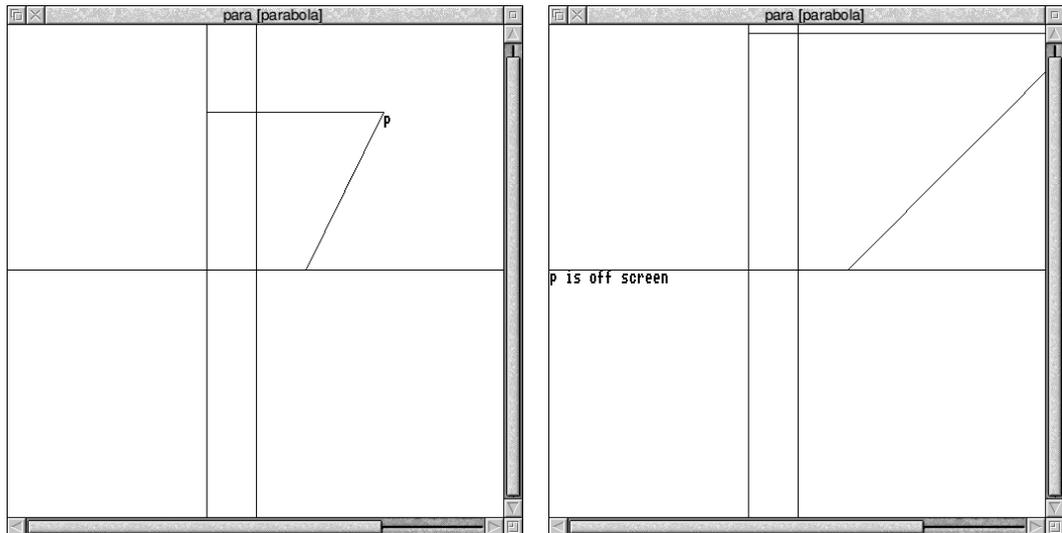


Figure 3.13: The “off screen” label in the parabola DAMscript

could be used. The following would be an excerpt from one possible solution.

```
parabola:      seti x0 500
dump:         print_store
parabola:      seti y0 500
parabola:      fif_add x x0 at2
...

```

Literal constants must be created in DAMscript. For example, line 39 of Listing 3.5 defines a symbol whose name is ‘0’ that has the literal integer value zero. This is similar to how literal bit patterns are encoded into instruction data streams in von Neumann processors: literal constants must be represented at some addressable location somewhere. The `seti` command on line 39 allows the remaining script to refer to the symbol 0 as the literal value zero. (Symbols following the named variable in a DAMscript *definition* are all interpreted as variable references, so there is no uncertainty whether in that context, 0 is a symbolic variable reference or a literal constant.) Unusually, the symbol 0 can be redefined at any time to take the value of any other integer (or indeed any 32 bit pattern — see the discussion of the type of variable `a` above), and all uses of the symbol 0 will then use the new value.

The final portion of the script (lines 62–82) demonstrate the use of boolean values and the `if` operator to create a warning label that is visible if the point `p` is off the screen, as shown in Figure 3.13.

```

warny = y0
smallwarnx = 0
bigwarnx = 1100

label { text = "p is off screen"
        y   = warny
        x   = if p < (0,0) or p > (2x0, 2y0)
              then smallwarnx
              else bigwarnx
      }

```

Listing 3.6: Definitions relating to the “off screen” label, written in a fictional language

The warning label is hidden when necessary by moving it to a coordinate that is beyond the far right of the visible screen. It is not possible to remove it completely in the current application for two reasons: 1) it is not possible to remove definitions in the DAM machine (although they can given some innocuous definition such as the literal value zero) and 2) as the location of graphical actions is not stored in the symbol table (see §3.3.3), no reference can be made to the relevant graphical action in order to remove it. The definitive construct in lines 62–82 that causes the label to be moved off-screen when appropriate could be represented in some fictional higher-level definitive language as shown in Listing 3.6.

The `if` construct is implemented here as a standard functional dependency using a conventional DAM machine operator. This implementation could be written out (in some other fictional, slightly lower-level definitive language) as:

```
x = if(cond, smallwarnx, bigwarnx)
```

The `if` construct has an unusual property that makes it a form of higher-order definition²⁶. In the example, the sources of `x` are the operator `if`, the boolean `cond` and *both* of the output cases `smallwarnx` and `bigwarnx`. Ideally, `x` should include *either* `smallwarnx` or `bigwarnx` in its sources, dependent upon the current value of `cond`. This would ensure that changes to the currently unselected value (`smallwarnx` or `bigwarnx`) would not cause the `if` definition (`x`) to be re-evaluated. The DAM machine does not implement higher-order dependency.

²⁶This appears to be the first time that the higher-order definition properties of the `if` construct have been explained in print.

3.4.3 DAM machine operators in BASIC and the dereference problem

In !Donald, new DAM machine operators must be written in ARM assembler, somewhat limiting the accessibility of the system. Also, some of the information about the script — notably the symbol table — is not easily accessible from ARM assembler as the information is maintained by code written in BASIC. These limitations are addressed below and another form of higher-order dependency is explained.

If a section of ARM assembler has been invoked using the BASIC CALL statement, it is possible for that assembler to call back into the BASIC interpreter, using the `EXPR` routine²⁷. In an attempt to overcome the limitations identified above, I implemented the new DAM machine operators `fn` and `gfn`, which take advantage of the `EXPR` routine, calling into the BASIC interpreter and allowing DAM machine operators and graphical actions to be coded in BASIC. Listing 3.7 below gives an example of their use in a DAMscript named “dereference”, and Listing 3.8 gives the text of the BASIC functions used.

```

1 # dereference DAMscript BASIC demo
2 # Ashley Ward (ashley@dcs.warwick.ac.uk) August 2002, December 2003
3
4 # standard definitions and literal values
5 seti a 1
6 seti b 2
7 iii_add c a b
8 iii_add d b b
9
10 # names of BASIC functions used by 'fn' and 'gfn' operators
11 setc varaddress "varaddress"
12 setc deref "deref"
13 setc plothex "plothex"
14
15 # symbolic names of definitions we are addressing
16 setc name1 "a"
17 setc name2 "b"
18 setc name3 "c"
19 setc name4 "d"
20
21 # addresses of values associated with each name
22 fn addr1 varaddress name1
23 fn addr2 varaddress name2

```

²⁷The `EXPR` routine is documented in [aco88, p.322]. It takes a tokenised BASIC expression string as input, evaluates it and returns the value. The location of the `EXPR` routine can be found through indirection of a value set in R14 by the BASIC CALL routine which invokes ARM assembler code. The precise implementation details are not relevant here — instead, this section focuses on the *use* of the facility.

3.4.3. DAM machine operators in BASIC and the dereference problem

```
24 fn addr3 varaddress name3
25 fn addr4 varaddress name4
26
27 # find values held at locations by dereferencing the addresses
28 # (should result in the value of each definition)
29 fn deref1 deref addr1
30 fn deref2 deref addr2
31 fn deref3 deref addr3
32 fn deref4 deref addr4
33
34 # describe geometry of table
35 seti xspace 200
36 seti yspace 40
37 seti x1 40
38 iii_add x2 x1 xspace
39 iii_add x3 x2 xspace
40 iii_add x4 x3 xspace
41 seti y1 980
42 iii_sub y2 y1 yspace
43 iii_sub y3 y2 yspace
44 iii_sub y4 y3 yspace
45 iii_sub y5 y4 yspace
46
47 # table column 1: symbolic names
48 setc nhead "NAME"
49 label x1 y1 nhead
50 label x1 y2 name1
51 label x1 y3 name2
52 label x1 y4 name3
53 label x1 y5 name4
54
55 # table column 2: values, referenced conventionally
56 setc vhead "VALUE"
57 label x2 y1 vhead
58 gfn plothex a x2 y2
59 gfn plothex b x2 y3
60 gfn plothex c x2 y4
61 gfn plothex d x2 y5
62
63 # table column 3: addresses
64 setc ahead "ADDR(NAME)"
65 label x3 y1 ahead
66 gfn plothex addr1 x3 y2
67 gfn plothex addr2 x3 y3
68 gfn plothex addr3 x3 y4
69 gfn plothex addr4 x3 y5
70
71 # table column 4: dereferenced values
72 setc dhead "DEREF(ADDR)"
73 label x4 y1 dhead
74 gfn plothex deref1 x4 y2
75 gfn plothex deref2 x4 y3
76 gfn plothex deref3 x4 y4
77 gfn plothex deref4 x4 y5
```

Listing 3.7: The “dereference” DAMscript

```

REM gfn "plothex" v x y
REM Plot a value v in hexadecimal at x, y
DEF FNplothex(v%,x%,y%,updatev%,xoff%,yoff%)
  IF updatev% = 1 THEN
    MOVE xoff% + x%, yoff% + y%
    PRINT ~v%
  ENDIF
= 0

REM fn resvar "deref" addr
REM Dereference a word of store at the given address
DEF FNderef(addr%, dummy1%, dummy2%, dummy3%, dummy4%, dummy5%)
= !(addr%)

REM Return the value of the null-terminated string at strptr%
DEF FNdamstring(strptr%)
  LOCAL r$, n%
  r$ = "": n% = 0
  WHILE (? (strptr% + n%) <> 0 AND n% <= 255)
    r$ += CHR$(? (strptr% + n%))
    n% += 1
  ENDWHILE
= r$

REM fn resvar "varaddress" varnamestr
REM Find the address of a named variable
DEF FNvaraddress(varnamestr%, dum1%, dum2%, dum3%, dum4%, dum5%)
= FNvariable_name_to_addr(FNdamstring(varnamestr%))
REM FNvariable_name_to_addr is a function defined in !Donald

```

Listing 3.8: The BASIC functions used by the Listing 3.7 DAMscript

Lines 34 onwards of the “dereference” DAMscript in Listing 3.7 define a graphical display in the form of a table. The numerical values in the table are defined using the `gfn` graphical action BASIC operator, using the BASIC function `plothex`. For example, the graphical action defined on line 58:

```
gfn plothex a x2 y2
```

results in a call to the BASIC function `plothex`, passing six arguments with the values of `a`, `x2`, `y2` and the “hidden” graphical action viewport variable and screen origin offsets. The BASIC function first checks that the viewport variable denotes

NAME	VALUE	ADDR(NAME)	DEREF(ADDR)
a	1	28024	1
b	2	28028	2
c	3	28068	3
d	4	2806C	4

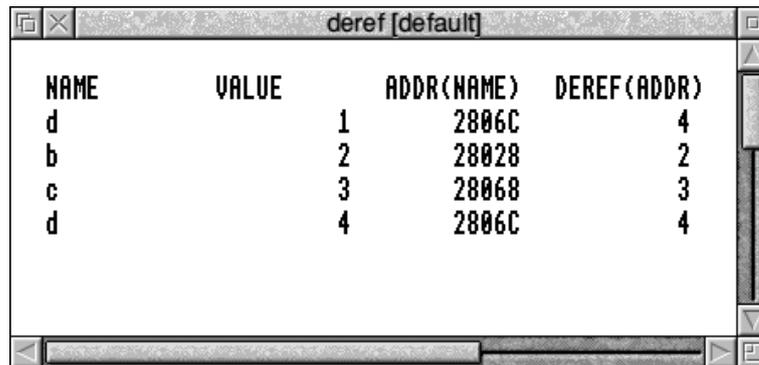
Figure 3.14: Graphical display from “Dereference” DAMscript, before and after a change to `yspace`

an active viewport (as this is a responsibility of all graphical actions — see §3.3.4); then moves the cursor to the position, correcting for the offset; and finally prints the passed value in hexadecimal.

The geometry of the table is defined in lines 34–45. When in its initial state, the graphical result is as shown on the left of Figure 3.14. If the value of `yspace` is changed from the initial 40 to 60, this change propagates to the target definitions in lines 42–45 and the table geometry is reconfigured. The result of this change is shown on the right of Figure 3.14.

As well as a demonstration of graphical table layout and use of BASIC operators, the “dereference” script is an attempt to use the symbol table (which is maintained by BASIC code) within the DAM machine. The initial part of the script (lines 5–8) defines a small set of values and definitions `a`, `b`, `c` and `d`. `a` and `b` are assigned literal values; `c` is defined to be the sum of `a` and `b`; `d` is defined to be $2*b^{28}$. The values of these definitions are shown in the second column of the table, by explicit reference to their symbolic names. The leftmost column of the table shows the values of `name1...name4`, which are initially set to `a...d`. The rightmost two columns of the table are defined in terms of the `name` values shown in the leftmost column. Specifically, the `ADDR(NAME)` column shows the addresses of the locations of the values associated with the names shown in the `NAME` column. The addresses are determined by using the `fn` operator to evaluate the BASIC `varaddress` function, which looks up the name using a BASIC function from the original !Donald. These addresses are then dereferenced to find the values at each address in the final `DEREF(ADDR)` column, using the `fn` operator on the BASIC `deref` function.

²⁸It could be redefined to be not $2*b$ — but that isn’t the question [Sha03].



NAME	VALUE	ADDR(NAME)	DEREF(ADDR)
d	1	2806C	4
b	2	28028	2
c	3	28068	3
d	4	2806C	4

Figure 3.15: “Dereference” after the change to name1

If we compose the definitions used, we can think of the values shown in the final column of Figure 3.14 as being calculated by the function:

```
DEREF(ADDR(NAME))
```

If we change one of the `name` variables, the resulting looked up value changes as expected. Figure 3.15 above shows the resulting display after the DAMscript command:

```
setc name1 "d"
```

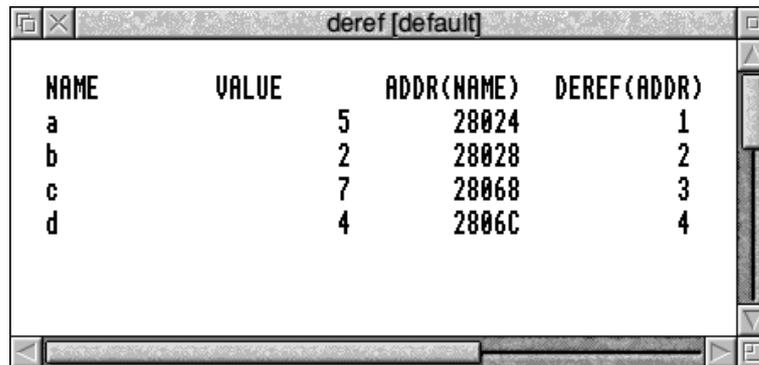
has been entered.

Notice that the top row of the last column correctly shows the current value of `d` (and also the third column shows the correct address). The second column still shows the value of `a`, as the graphical actions in the second column make explicit reference to symbolic names. The propagation of change invoked by changing a `name` variable is therefore correct.

However, the implementation of the indirect referencing means that some necessary propagation of change is omitted. If the value of `a` is redefined to 5, using the following DAMscript command:

```
seti a 5
```

then the resulting graphical display is as shown in Figure 3.16.



NAME	VALUE	ADDR(NAME)	DEREF(ADDR)
a	5	28024	1
b	2	28028	2
c	7	28068	3
d	4	2806C	4

Figure 3.16: “Dereference” after the change to a

The values shown in the second column of Figure 3.16 (via an explicit reference) are all consistent with their definition:

$$c = 5 + 2 = 7$$

for example. However, the values in the last column have not changed from the pre-change configuration (which is shown in Figure 3.14). This is because the definition values visible in the last column only *indirectly* include a as a source. When the value of a changes, the value of:

DEREF(ADDR("a"))

is not updated. This flaw in the implementation leaves values inconsistent with their definition in this circumstance and exposes values from the previous state, before the transition was made.

Stated more abstractly, the sources of the composed function:

DEREF(ADDR(s))

should include the symbol whose name is referenced by the string **s**. If the string **s** changes, the sources of the function should change²⁹. This causes a problem as it violates a fundamental assumption made in most definition maintainers: that the script graph only changes when a definition is changed, not when a literal value is changed.

²⁹The same issue is encountered in spreadsheet tools that include the INDIRECT function — see p.33.

The symbol table is thus a kind of Higher Order Definition. It is comprised of symbolic references to data. When a definition uses a symbolic reference, the definition should be updated whenever the data changes as well as when the reference changes. This is not easily possible within the current DAM machine.

This problem can be related to the problem with the `if` operator described at the end of the previous section §3.4.2. There, the sources of:

```
if(cond, a, b)
```

changed according to the value of `cond`. Again, the script graph is changing when a literal value is changed.

The two examples show the need for a HOD facility in the definition maintainer, where definitions can modify the script graph. It is possible to envisage a DAM machine operator (perhaps implemented in BASIC in `!Donald2`) which calls `addtoq` in order to change a dependency. This might be termed a dependency-modifying action as, similar to a graphical action, it performs some action as a side effect of its invocation. The scheduling problems discussed at the end of section §3.3.4 in the context of graphical actions are then relevant to this problem. The dependency-modifying actions must be performed first in the update algorithm, as their action affects the script graph and hence the topological sort and ordering of operator invocation in phase 2 of the BRA.

3.4.4 **!Donald performance**

Cartwright's thesis presents some timings that compare `!Donald` and `tkeden`³⁰ and makes some preliminary claims about the advantageous performance of `!Donald`. This thesis explores both `!Donald` and EDEN in some depth, so a comparison would seem welcome. The purpose of this section is to revisit Cartwright's performance evaluation and explore the issues further. Does a definition maintainer using the BRA and coded in low-level assembler perform better than EDEN, coded in C and using a complex scheduling algorithm described in §4.3?

Cartwright evaluates the performance of two DoNaLD scripts in both `!Donald` and `tkeden`. The two scripts are the engine script (pictured in Figure 3.6, p.126) and

³⁰Around 1999: before I introduced a full version numbering scheme.

a “shapes” test script. Both scripts are DoNaLD, not DAMscript. Measurements were taken by repeatedly increasing a parameter chosen to cause much screen geometry to be modified on each change, using the Update Variable facility in !Donald and a small script in `tkeden`, and recording the total time taken to move through all the states. !Donald was run on the Acorn platform described in §3.2.1 and `tkeden` on a SPARCstation 2. The following is an excerpt from [Car99, §5.4.2].

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.

... 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. 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`...

The table [Table 3.5] shows how the multi-tasking versions of the animation of the engine script produced a factor of 10 improvement³¹ 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.

Cartwright’s table is reproduced as Table 3.5³². Timings are totals shown in seconds — lower figures are better.

The first sentence of the quoted text does acknowledge that the performance results are a demonstration of potential only. The argument presented there starts from the assumption that the two machines used have similar performance, as evidenced by similar MIPS ratings, and so timings can be compared. Unfortunately MIPS is not a good basis on which to make this comparison, as it assumes a comparable amount of work is done in each processor instruction. Patterson and Hennessy [PH98, p.76] state:

There are three problems with using MIPS as a measure for comparing machines. First, MIPS specifies the instruction execution rate but does not take into account the capabilities of the instructions. We cannot compare computers

³¹The “factor of ten improvement” claim also appears in [ABCY98].

³²The table headings have been changed, the column ordering organised and multipliers added to indicate performance relative to preceding columns, in the same manner as the quoted text.

Script	<code>tkeden</code> on a SPARCstation 2	!Donald on the Acorn, multi-tasking GUI mode	!Donald on the Acorn, single-tasking GUI mode
<i>Engine</i>	262	26 (x 10 over <code>tkeden</code>)	13 (x 20 over <code>tkeden</code> , x 2.0 over !Donald GUI)
<i>Shapes</i>	172	21 (x 8 over <code>tkeden</code>)	15 (x 11.5 over <code>tkeden</code> , x 1.4 over !Donald GUI)

Table 3.5: `tkeden` and !Donald performance compared when running on different machines (from [Car99, p.169])

Script	<code>tkeden</code> on the Acorn under ARM Linux	!Donald on the Acorn under RISC OS, multi-tasking GUI mode	!Donald on the Acorn under RISC OS, single-tasking mode
<i>Engine</i>	530	30 (x 18 over <code>tkeden</code>)	13 (x 41 over <code>tkeden</code> , x 2.3 over !Donald GUI)
<i>Shapes</i>	650	53 (x 12 over <code>tkeden</code>)	35 (x 19 over <code>tkeden</code> , x 1.5 over !Donald GUI)

Table 3.6: `tkeden` and !Donald performance compared when running on the *same* machine

with different instruction sets using MIPS, since the instruction counts will certainly differ. Second, MIPS varies between programs on the same computer; thus a machine cannot have a single MIPS rating for all programs. Finally and most importantly, MIPS can vary inversely with performance!

Running all tests on the same machine would remove the hardware variable from the comparison. Accordingly, this author installed ARM Linux (version 2.0.36) on the Risc PC and compiled `tkeden` (version 1.17). Cartwright’s tests were recreated as far as possible³³, all running on the Acorn machine. The test results are shown in Table 3.6.

Although the absolute timings are different, the ratios of improvement that

³³Some information about the original tests is no longer available — particularly, the range over which the variable was changed and the step size. In the engine script, the `/lup` variable, and in the shapes script, the `/update` variable were changed from 0 to 255 in integer steps (although in the case of the engine script, the variable is actually of floating type type).

single-tasking mode has over multi-tasking mode are the same as in Cartwright's results, giving confidence that this is a similar experiment. However, the performance of `tkeden` appears significantly worse on the Acorn as compared to the SPARC-station 2, despite the Acorn having a higher MIPS rating than the SPARC, again indicating that MIPS is not a useful basis for comparison here. Is the DAM machine at least a 12 fold improvement over `tkeden`, then, or is `tkeden` just performing badly under the ARM Linux OS?

Much of the difference in observed total time may be due to the differing graphical hardware employed on each platform. Running `tkeden` on the same hardware as !Donald eliminates that source of variation, but there are major differences in the graphical subsystems *software* architecture between RISC OS and ARM Linux and in the ways that !Donald and `tkeden` use these. For example:

- The OS graphical routines called by !Donald on the Acorn write directly to screen memory, whereas the Tcl/Tk graphical routines used by `tkeden` on UNIX platforms communicate with an X Windows server process which writes to screen memory, using the X Windows protocol, facilitating graphical display from remotely executing programs.
- Qualitatively, Cartwright points out that the Acorn seems to produce smoother animation than `tkeden`. This is true, and is due to the use of a double-buffering technique employed by Alderidge in !Donald. The complete new state is drawn into an off-screen buffer which is then made visible during the CRT flyback period (thus limiting the frame rate to 60Hz in the single-tasking screen mode). The drawing process is thus not visible to the external viewer. I added an option in !Donald2 to disable the use of double-buffering, and the engine script running in this mode does not update smoothly. However it completes the above experiment in the even faster 8 seconds (*cf.* 13 seconds in Table 3.6), as in this mode it no longer needs to wait for the CRT flyback period. This is an example where the prediction of major transition time discussed on p.104 would prove useful — if this were possible, the ability of the machine to meet the real-time deadlines imposed by the CRT flyback period could be guaranteed for a particular definitive script.

Eden	DAMscript
<code>i=1;</code>	<code>seti i 1</code>
<code>i_sqr is i*i;</code>	<code>i_sqr i_sqr i</code>
<code>iii_add is i+i_sqr;</code>	<code>iii_add iii_add i i_sqr</code>
<code>iii_sub is i_sqr-iii_add;</code>	<code>iii_sub iii_sub i_sqr iii_add</code>
<code>iii_mult is iii_add*iii_sub;</code>	<code>iii_mult iii_mult iii_add iii_sub</code>
<code>iii_div is iii_sub/iii_mult;</code>	<code>iii_div iii_div iii_sub iii_mult</code>
<code>f_int is float(i);</code>	<code>f_int f_int i</code>
<code>f_sqr is pow(f_int,2.0);</code>	<code>f_sqr f_sqr f_int</code>
<code>fff_add is f_int+f_sqr;</code>	<code>fff_add fff_add f_int f_sqr</code>
<code>fff_sub is f_sqr-fff_add;</code>	<code>fff_sub fff_sub f_sqr fff_add</code>
<code>fff_mult is fff_add*fff_sub;</code>	<code>fff_mult fff_mult fff_add fff_sub</code>
<code>fff_div is fff_sub/fff_mult;</code>	<code>fff_div fff_div fff_sub fff_mult</code>
<code>i_flt is int(fff_div);</code>	<code>i_flt i_flt fff_div</code>
<code>bii_eq is iii_div==i_flt;</code>	<code>bii_eq bii_eq iii_div i_flt</code>

Listing 3.9: “Numeric” Eden script and DAMscript

- !Donald draws the complete new state after every transition, as all the graphical actions in the currently active viewport are invoked. In comparison, `tkeden` communicates only the changed geometry to Tcl/Tk, which is then responsible for moving the current display into the new state by un-drawing changed old state and drawing the new state.

Cartwright points out that the “shapes” script is likely to have a higher trigonometry calculation : line drawing ratio than the “engine” script. The lower improvement ratio that the DAM machine shows for the “shapes” script, both over `tkeden` and in single-tasking mode improvement over GUI mode, may be due to the larger amount of basic value calculation during update. To test this, this author constructed the “numeric” Eden script and DAMscript shown in Listing 3.9.

The two equivalent scripts calculate numeric values only: there are no graphical actions. Each script combines some operators into a graph structure. The same structure is built using integer operators and also using floating point operators, the two structures being linked by the single input and single output which gives a boolean result. The structure of the script is illustrated in Figure 3.17.

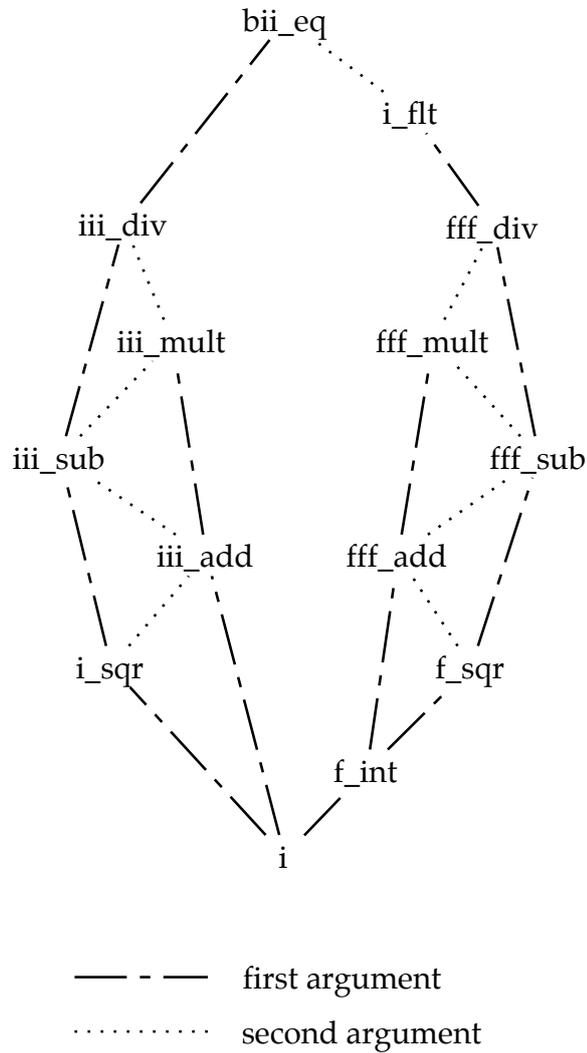


Figure 3.17: Structure of the “numeric” script

Script	<code>tkeden</code> on the Acorn under ARM Linux	<code>!Donald2</code> on the Acorn under RISC OS, single-tasking mode
<code>numeric</code>	29 secs	26 secs (x 1.1 over <code>tkeden</code>)

Table 3.7: `tkeden` and `!Donald2` performance when running the “numeric” non-graphical script (Listing 3.9)

The variable `i` was changed from 1 to 10240 and the total time taken to complete the move through all the intermediate states was recorded. The performance of the two definition maintainers is then very similar, and is shown in Table 3.7.

The performance advantage that the DAM machine appears to give may therefore be almost entirely due to efficient graphical routine implementations in the RISC OS operating system. (The slightly degraded performance of `tkeden` here may be due to kernel overheads imposed in a pre-emptive UNIX process scheduler, compared to the RISC OS cooperative scheduling system, where there are no other tasks running simultaneously with `!Donald2` running in single-tasking mode.)

The major performance feature of the DAM machine — the BRA facility to optimise the update resulting from a block of multiple redefinitions — is not measured in the above tests, as only one variable was repeatedly changed. It is in fact rather difficult to test this facility in the current versions of `!Donald` as the user interfaces do not permit more than one redefinition to be made simultaneously. Chapter 4 contains an evaluation of EDEN in the light of the theoretical advantages of the BRA. It is found that when Eden's `autocalc` facility is used to demarcate blocks of redefinitions, the total number of evaluations performed is the same as that that the BRA would produce.

In definitive systems, as in many other areas of computing, then, performance comparisons are difficult to make. Performance comparisons in non-definitive systems are often made with respect to a “benchmark”, which contains a particular mix of instructions relevant to some domain. Similarly, the evaluations above show that the performance of a definitive system is largely dependent upon the mix of operators used in the script. `!Donald` performs well if the script has a large proportion of graphical actions (in keeping with its principal function) but it gives no significant advantage over `tkeden` when the script involves only floating point and integer values with no visual representation. As there is no such thing as the average program, there is no such thing as the average script, making any attempt at a general performance comparison impossible.

3.5 Geometry of definitions in the DAM machine store

Cartwright’s DMM and DAM machine design both abstract away the actual *location* of maintained machine words. In !Donald, this issue is addressed in the symbol table implementation, which is implemented as a part of !Donald using a mix of BASIC and ARM assembler. We have seen in §3.4.3 that the indirection introduced by a symbol table introduces problems of higher-order dependency. This section shows:

- that the abstraction causes problems when using dependency on data items whose representation is larger than one word or of variable size;
- how the geometry of definitions in store can be important when dependency is applied at a finely granular level for visualisation;
- how the abstraction moves definitive scripts away from the spatial aspects of spreadsheets, highlighting respects in which it is inappropriate to view definitive scripts as “generalised spreadsheets” (a description given, for example, in [Bey90]);
- and how the video hardware subsystem in the Acorn platform can be considered to be performing definition maintenance as a part of the DAM machine.

3.5.1 Dependency between multi-word data types

The DAM machine maintains dependencies between single words in store only. Provided that every data type used can be completely represented in a 32-bit word, this at first seems adequate. The conventional thinking would be that any problems that arise from this limitation can be dealt with at higher levels in the architecture. For example, the DAM machine provides no facilities for identifying the data type of a word, so careful use of operators must be made to ensure that the correct operators are used on the correct data type. It seems that this can be done adequately well at a higher level in most applications.

When a multi-word data representation is to be used, Cartwright suggests creating an operator for each 32-bit word component of the result. Figure 3.18 illustrates

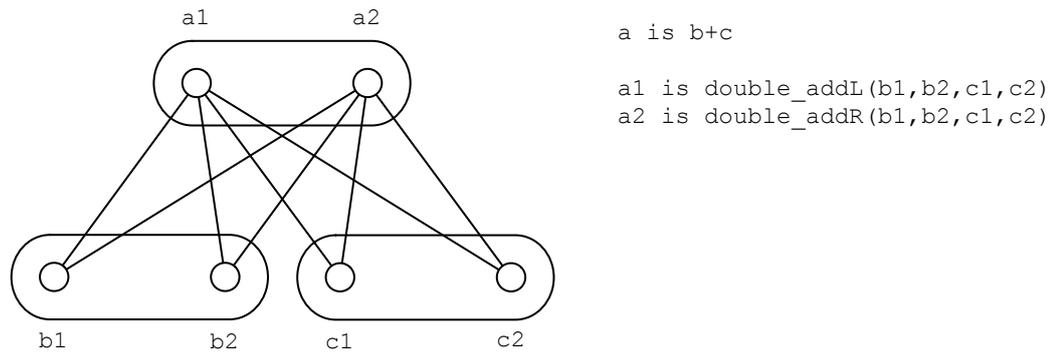


Figure 3.18: DAM machine operator configuration for summation with double length data types

the idea. The figure is based on [Car99, p.158]³⁴. In Figure 3.18, *a*, *b* and *c* are 64-bit values, which are each split into two 32-bit word components. Splitting multi-word operators into multiple 32-bit operators increases the number of operators required and the number of nodes and arcs in the script graph. Also notice that each `double_add` operator requires all four arguments, due to the potential of a carry bit from the least significant word during the addition. Much of the addition operation will thus have to be performed twice, reducing performance.

3.5.2 Dependency on variable length data

Notice that the !Donald string data type is not represented within the current DAM machine store: the store contains only a 32-bit pointer to the string, which is stored elsewhere, as illustrated by `'intervar9'` and `'intervar15'` in the previous Figure 3.10 (p.136 — a graphical representation of the Listing 3.3 DAMscript on p.133, itself a result of translation from the Listing 3.2 DoNaLD script).

The splitting-operator solution can only be applied to fixed length data types. If the variables *b* and *c* in Figure 3.18 were variable length data types, the dependency links between them and the variable *a* would need to be modified when their length changed. The operators involved would also need to be changed to cope with the varied number of arguments.

³⁴But with identifiers changed to make clearer that *a1* and *a2* are components of *a* and with the corresponding script included.

```

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

```

Listing 3.10: A DAM machine ‘lookup’ operator (from [Car99, p.159])

For arrays, Cartwright suggests the use of a `lookup` operator in order to find the q th element in an array whose base address is p , giving the code shown in Listing 3.10 (pre- and post-requisites added by this author).

Unfortunately this solution has the same problem discussed in the context of the `DEREF()` and `if` operators at the end of section §3.4.3. Suppose that a definition `val` is created which uses Cartwright’s `lookup` operator in the following hypothetical DAMscript:

```

seti arraybase &27D3435
seti offset 3
lookup val arraybase offset

```

Initially, `val` will have the expected value, taken from the memory location `27D34` (hex) + $4*3$, but when the data at that location changes, the value of `val` will not be automatically updated. This operator and its use within the DAM machine does not capture all the dependency present in this situation.

One solution to this problem would be to explicitly create all the necessary dependency. With an imagined extension to DAMscript, using the `!` character to denote explicit reference to memory locations rather than the symbol table, an

³⁵The `&` prefix denotes a hexadecimal value in BBC BASIC. The example address used here is hypothetical — it would be difficult to determine the necessary addresses in the current `!Donald2` application.

example might look like the following:

```
seti arraybase &27D34
seti offset 3
lookup val arraybase offset !&27D34 !&27D38 !&27D3C !&27D40
```

Here, the elements of a four word array have been added as explicit arguments to the `lookup` operator, ensuring that changes to data will cause `val` to be recalculated. However, the total number of arguments is limited to 10 in the DAM machine (since operator arguments are passed using processor registers), so this is not a general solution. Even if more arguments were possible, this solution creates a large amount of data in the Sources Store for `val`, and the same amount of data (but spread across the entire list) in the Targets Store.

Also, the solution is now over-prescribing the dependency present in the situation. If the second element in the array (at location `&27D38`) is changed, `val` will be recalculated, although it is currently referring to the third element. The value of `val` is actually dependent only on the value at location `(arraybase + offset)`. We have once again encountered the Higher Order Definition problem of the script graph changing when literal data changes.

One of the causes of this problem is the fact that the DAM machine was designed to maintain dependencies within a *set* of values. The set is ordered only by the script graph, which represents the dependency relationships over a set of value identities. The value identities themselves are not themselves comparable. This corresponds to the mathematical graph abstraction: only the set of nodes and the set of edges between nodes is relevant — the geometrical positions of a node (and hence the graphical layout of the graph) is irrelevant. In the DAM machine, the choice of where in store to place a word has no discernible effect on the working of the machine. Similarly, in a definitive script, changing the symbolic name of a definition and all references to that definition has no effect.

Within present EM tools, the problem of dependency maintenance has been abstracted away from the geometrical layout of the store. This abstraction can be considered a positive feature, as it represents a separation of concerns [Dij76, p.211], but with the abstraction come limitations of the sort described with the `lookup` operator. Arrays and lists imply geometrical relationships between data elements — if there is no geometry, then arrays and lists are difficult to implement

(e.g. linked lists must be employed). The abstraction has therefore impoverished the kinds of reference that can be employed. The contrast with dependency maintenance in spreadsheet tools is instructive in this context. The grid layout employed in spreadsheet tools does not just enable a spatially grounded user interface — the tools also exploit the geometrical relationships between cells to provide various forms of local referencing. A cell can reference “the cell above this one”, and the cell can then be replicated (copied with formulae substitution) to another location, preserving the local reference.

3.5.3 Visualising the store

Despite the abstraction, the store in !Donald2 of course exists in a particular region of memory and is ordered (as is all other data in the machine) by address. As an experiment, I have added a facility for visualising the store directly³⁶ to !Donald2. This was done by allocating an extra RAM screen buffer and locating the DAM machine store within it. When running in single-tasking mode, a key can be pressed, causing the video hardware to be reconfigured to display the DAM store screen buffer. The results for the parabola script (Listing 3.5, p.144), which contains 38 definitions and the larger engine DoNaLD script (shown running in Figure 3.6, p.126), which contains approximately 800 definitions, are shown in Figures 3.19 and 3.20 respectively. (Note that the screen shots are colour inverted for this thesis as black-on-white prints more acceptably.)

In the video mode used for the visualisation, the screen is 640x480 pixels with a bit depth of 1 (i.e. each pixel can be either black or white). The screen is addressed starting at the top and moving along each row from left to right (corresponding to the raster scan of the hardware). Each DAM machine word is 32 bits and corresponds to 32 laterally adjacent pixels, least significant bit left-most. Each screen row shows 20 words. The screen has 480 rows, showing 9600 words in total. The top 96 rows show the 1920 maintained DAM word values. The remaining 384 rows (4/5ths of the total) show the Knuth Counter, Function, Target and Source Pointers respectively.

³⁶[Car99, §5.5] claims to have achieved something similar, but the effect is actually achieved indirectly: after the BRA update has completed, some code (written in BASIC) copies a portion of the DAM store state onto the screen, making the appropriate geometrical mapping to pixel data as it goes.

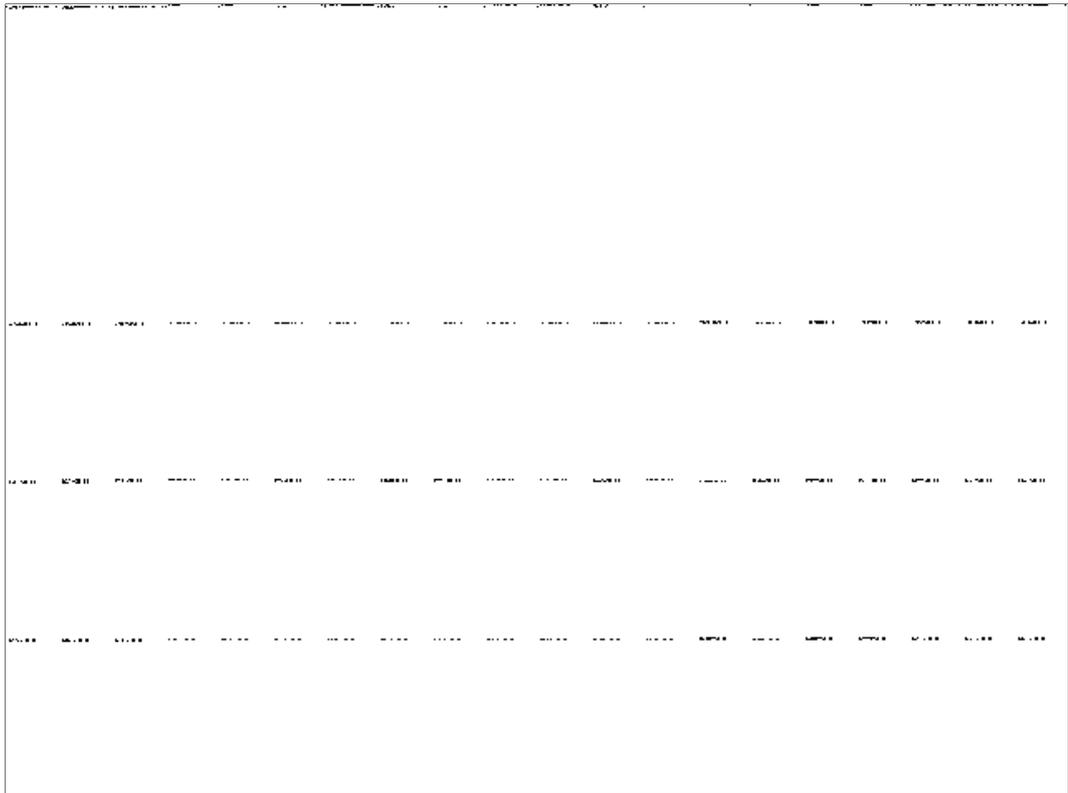


Figure 3.19: Visualisation of the “parabola” DAMscript DAM store

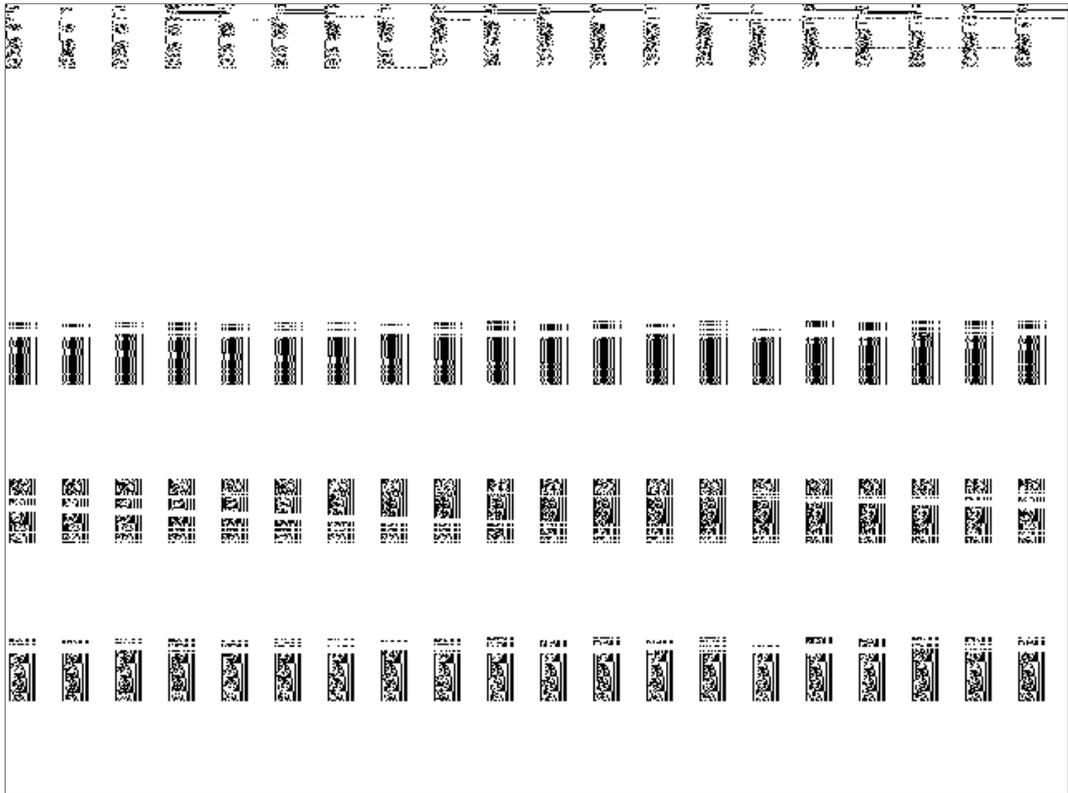


Figure 3.20: Visualisation of the “engine” DoNaLD script DAM store

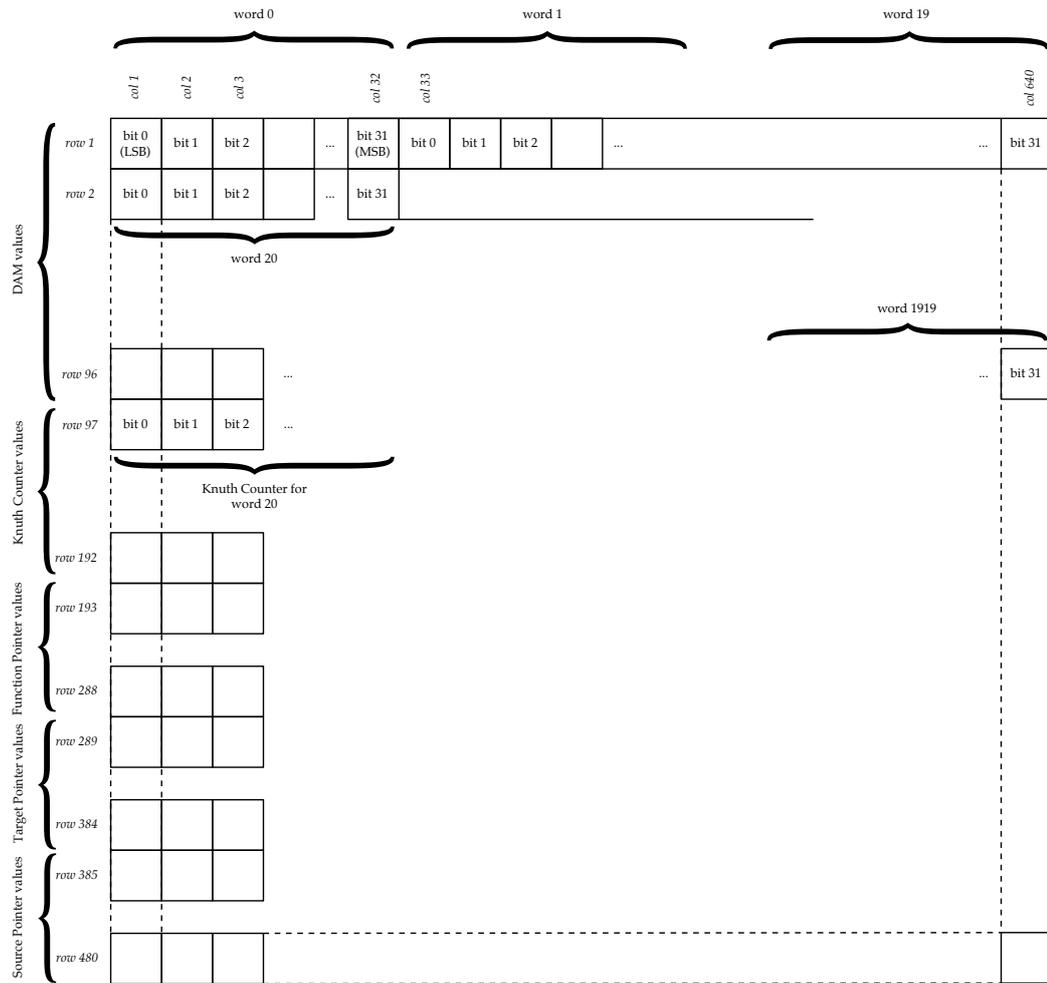


Figure 3.21: DAM store mapping to pixels

The pixel display therefore corresponds to a geometrical representation of some of the DAM machine data structure (which was shown in Figure 3.2 on p.115). The geometrical layout of the pixels is shown in Figure 3.21.

The “parabola” and “engine” examples shown use mostly byte size types, so only the 8 low order bits in each 32-bit word are used. This explains the large amounts of white space between each word in the Figures 3.19 and 3.20.

As the values in the script change in response to a single change entered in DAMscript or a sequence of automated changes from the Update Variable window, the pixels on the screen that show targets of the change can be observed to change. The Knuth Counter values are not visible in the screenshots, as the counters take

non-zero values only when the `update` routine is running, and whilst the routine is running it is not possible to capture a screenshot. However, the values are visible on screen whilst a `update` is in progress.

We can therefore consider the video hardware to be establishing a visual dependency for us. The brightness of the top left pixel in the display *is* the status of the least significant bit in the first DAM machine word. Due to this combination of video hardware and software definition maintainer, it is possible to answer the question “why is that pixel white?” without reference to past history of sequential code execution — it is white because it is defined to be that way, and the dependency structure can be traced to determine the values that affect the pixel brightness.

3.5.4 Exploiting the visualisation

This final section shows how control over the geometrical position of a given definition was obtained in `!Donald2` and how this was then exploited to produce a pixelated display with meaningful state.

`!Donald2` does (of course) deterministically map words to particular locations. The order that commands are given in a DAMscript file has an effect on the mapping to locations. As the DAMscript file is interpreted in two passes, the mapping is rather complex. DAMscript definitions and values are mapped to locations starting from the smallest numbered location in the DAM machine store, in the following order.

1. The default viewport variable, the x offset and the y offset (3 words in total) are created.
2. ‘set’ values, in the order presented, are mapped.
3. Dependencies, and a variable for each `viewport` command, are mapped, in the order presented. For a definition *operator_name variable dependent_list*:
 - (a) First the references in *dependent_list* (left to right) are mapped, unless already defined,
 - (b) Then the *variable* is mapped.

After having analysed this mapping, I exploited it in a rudimentary way. The “text” DAMscript shown in Listing 3.11 is ordered such that the loading process maps the values to DAM machine store in the same order that they are shown in the script — i.e. the mapping transformation described above has no effect. Listing 3.11 is an excerpt from the full listing, showing the first three rows only.

```
1 # text DAMscript example
2 # Ashley Ward (ashley@dcs.warwick.ac.uk) December 2003
3
4 # This script is ordered in the same way that it will be loaded
5 # and hence geometrically row 1 first, L to R, then row 2...
6
7 # -- ROW 1 follows -----
8
9 # col 01: taken by default viewport variable
10 # col 02: taken by viewport x offset
11 # col 03: taken by viewport y offset
12
13 # col 04: character code 1
14 seti cc1 65
15 # col 05: character code 2
16 seti cc2 115
17 # col 06: character code 3
18 seti cc3 104
19
20 # col 07: literal
21 seti zero 0
22 # col 08: literal
23 seti one 1
24 # col 09: literal
25 seti two 2
26 # col 10: literal
27 seti three 3
28 # col 11: literal
29 seti four 4
30 # col 12: literal
31 seti five 5
32 # col 13: literal
33 seti six 6
34 # col 14: literal
35 seti seven 7
36 # col 15: literal
37 seti eight 8
38 # col 16: pointer to name of BASIC function used
39 setc chargraphic "chargraphic"
40 # col 17: value shown in unused locations
41 seti blank 255
42 # col 18: fill to end of row
43 seti r1c18 0
44 # col 19: fill to end of row
45 seti r1c19 0
46 # col 20: fill to end of row
47 seti r1c20 0
```

```
48
49 # -- ROW 2 follows -----
50
51 # col 01: dummy blank
52 iii_add r2c01 zero blank
53 # col 02: dummy blank
54 iii_add r2c02 zero blank
55
56 # col 03: row number
57 iii_add rn1 zero one
58 # col 04: character graphic 1
59 fn r2cg1 chargraphic cc1 rn1
60 # col 05: character graphic 2
61 fn r2cg2 chargraphic cc2 rn1
62 # col 06: character graphic 3
63 fn r2cg3 chargraphic cc3 rn1
64
65 # col 07..20: dummy blank
66 iii_add r2c07 zero blank
67 iii_add r2c08 zero blank
68 iii_add r2c09 zero blank
69 iii_add r2c10 zero blank
70 iii_add r2c11 zero blank
71 iii_add r2c12 zero blank
72 iii_add r2c13 zero blank
73 iii_add r2c14 zero blank
74 iii_add r2c15 zero blank
75 iii_add r2c16 zero blank
76 iii_add r2c17 zero blank
77 iii_add r2c18 zero blank
78 iii_add r2c19 zero blank
79 iii_add r2c20 zero blank
80
81 # -- ROW 3 follows -----
82
83 # col 01: dummy blank
84 iii_add r3c01 zero blank
85 # col 02: dummy blank
86 iii_add r3c02 zero blank
87
88 # col 03: row number
89 iii_add rn2 zero two
90 # col 04: character graphic 1
91 fn r3cg1 chargraphic cc1 rn2
92 # col 05: character graphic 2
93 fn r3cg2 chargraphic cc2 rn2
94 # col 06: character graphic 3
95 fn r3cg3 chargraphic cc3 rn2
96
97 # col 07..20: dummy blank
98 iii_add r3c07 zero blank
99 iii_add r3c08 zero blank
100 iii_add r3c09 zero blank
101 # (etc) ...
```

Listing 3.11: “text” DAMscript

```

REM fn resvar "chargraphic" charcode rowno
REM Find the character graphic (one row only)
REM for a particular character code
DEF FNchargraphic(charcode%, rowno%, d3%, d4%, d5%, d6%)
  DIM chargraphicblock% 8
  chargraphicblock%?0 = charcode%
  SYS "OS_Word", 10, chargraphicblock%
  = chargraphicblock%?rowno%

```

Listing 3.12: chargraphic BASIC function used by “text” DAMscript (Listing 3.11)

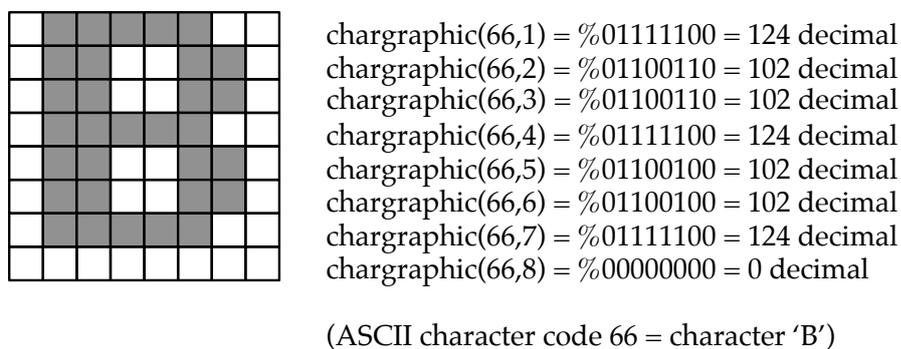


Figure 3.22: Lookup of an ASCII character code glyph

The DAMscript creates three character patterns on the screen. The pixels showing the character patterns are dependencies of the form:

```
fn r2cg1 chargraphic cc1 rn1
```

i.e. a variable named `r2cg1` which uses the BASIC function `chargraphic` on the arguments `cc1` and `rn1`. The `chargraphic` BASIC function is shown in Listing 3.12. The function takes an ASCII character code and a row number (from 1 to 8) as input. It queries the 8x8 pixel character glyph graphic maps maintained by the operating system and returns the row of pixels for the given character and row as a byte-wide result, as illustrated in Figure 3.22.

The character pattern dependencies are shown in Listing 3.11 on lines 59–63 and 91–95. There are six other similar regions of definitions (making up the six rows of pixels not described in Listing 3.11) in the full listing. The character pattern dependencies have as sources the word at top row of their column and the word in col 3 of their row, holding the ASCII value and the row number (lines 14–18 and

lines 57, 89 and six other similar definitions respectively).

The screen pixels are laid out on the screen in a grid form, and so the dependency present in the script resembles a spreadsheet. Correspondingly, Figure 3.23 shows a spreadsheet with cell formulae set to similar dependencies as created in the “text” DAMscript.

The location transformation during the loading process forces definitions representing literal values to the earlier words in the DAM store. In this script, the first row is formed entirely from literal values. Hence, the row numbers (which are dispersed among the later dependencies) are formed by simple dependencies which add zero to the desired value. This is illustrated in the spreadsheet shown in Figure 3.23. Words that are unused, but which must be defined in order to coerce the loading process into creating the character patterns in the correct positions, are defined as equal to the “blank” value at row 1, col 17 (line 41 in Listing 3.11). The spreadsheet shown in Figure 3.23 does not show these definitions.

When the DAMscript is loaded into !Donald2 and the store visualisation is turned on, the resulting display is as shown in the lower part of Figure 3.24. The upper part of the figure shows a magnified portion of the upper left corner of the screen. The three character patterns for the letters **A**, **s** and **h** (ASCII 65, 115 and 104) are clearly visible at the top left. The patterns are laterally reversed, due to a mismatch between the output of the `chargraphic` function and the memory to pixel mapping described in Figure 3.21. The words showing the ASCII codes are also discernible at the top of each character pattern column, as are the row numbers to the left of the **A** pattern. The “blank” value is set to 255, creating the solid eight-pixel wide blocks at unused positions.

“Why is that pixel white?” is now a question whose answer can be traced from pixel location through to definition and determining sources. Changing the ASCII values or the row numbering changes the patterns shown on the screen by dependency.

The “text” DAMscript and !Donald2 configuration of video hardware is the first step towards an entirely definitive text editor (compare Y.W. Yung’s first Eden text editor model described in §4.1.3), representing the output side of such an implementation. !Donald2 does not yet provide definitive *input* facilities, but this is

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1																				
2																				
3																				
4																				

Figure 3.23: Spreadsheet form of “text” DAMscript

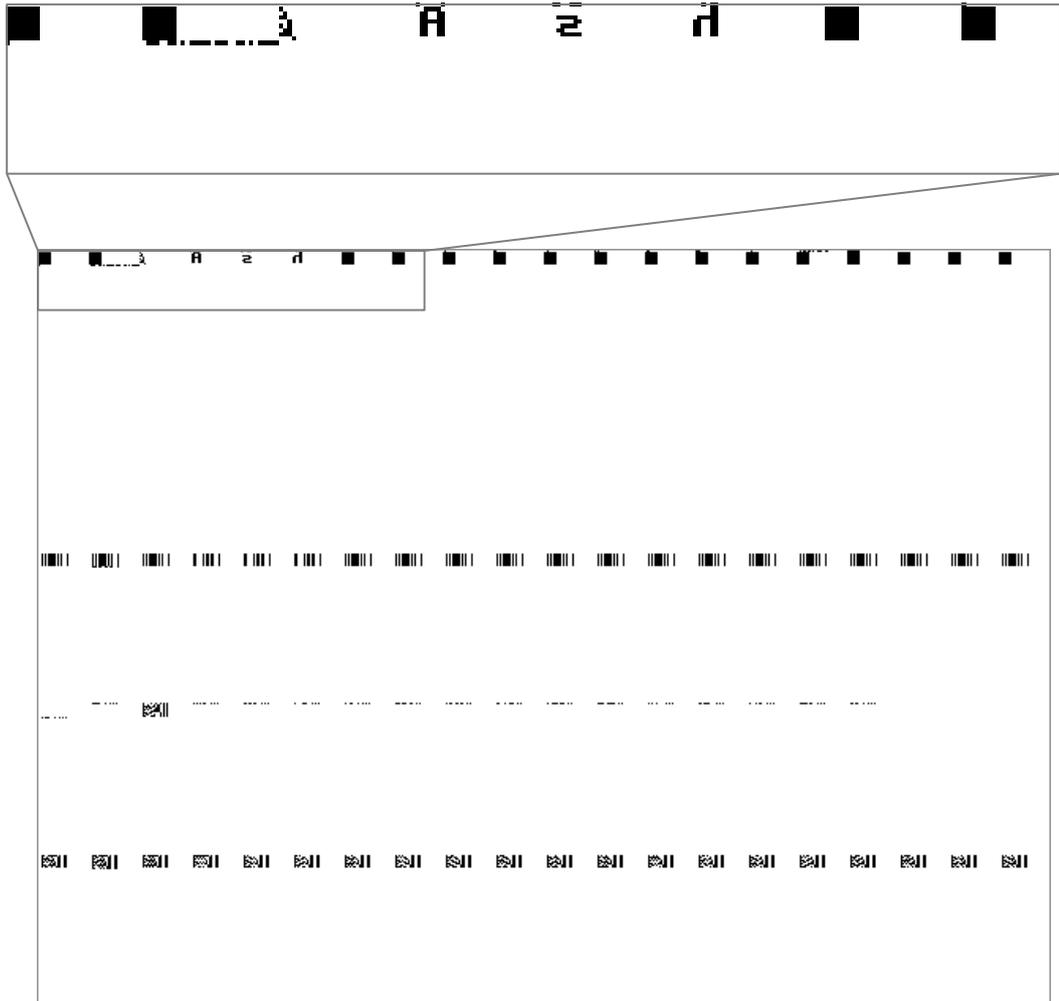


Figure 3.24: Visualisation of the “text” DoNaLD script DAM store

possible. If this were implemented, the determining sources of a pixel at the current cursor position might include the keyboard input. The operation of the machine could then be construed in a similar way to a mechanical typewriter and be analysed along similar lines, with definitions providing hard guarantees of cause and effect, like levers in the mechanical counterpart.

A full screen representation based on the principles illustrated would require several thousand³⁷ dependencies operating at a finely granular level. Exploiting dependency in this way is made more difficult by abstracting away the geometrical layout of store, so that identities cannot be compared with one another. The two-dimensional geometrical representation is appropriate for the screen display, but other geometrical representations might be more appropriate for other applications. In principle, different geometric representations might be used to organise different regions within the same machine store.

³⁷9600 to be precise — see p.166.

3.A The Script Digraph

Dependency describes a relationship between the variables in a definitive script. This section links this notion to basic graph theory, describing the structure I call a *script digraph*, or more informally, a *script graph*.

We can draw a diagram representing the dependency in a definitive script. Conventionally, we draw a graph structure. First, we represent each variable (definition or literal value) by a diagram point, or *node*. Secondly, we draw a line to show each use of a variable value. A line is drawn between a definition (say *d*) and a variable (say *v*) whose value is used by that definition. Because a change to the value of *v* implies that the value of *d* will change, we indicate a direction on the line, making it an arrow, or *arc*. This makes the whole structure a *directed graph*. Conventionally, we draw the direction of the arrow to show the *effect* of change: i.e. from *v* to *d*. Interpreted in the reverse direction, the arc shows a possible *cause* of change.

A directed line is required as there is no symmetry in the relationship between *d* and *v*: although a change to *v* implies a change to *d*, a change to *d* does not imply a change to *v*. This distinguishes dependency from traditional constraints.

An arc is drawn from every use of a variable to the definition that uses it. If a definition uses the same value twice, we only draw one arc — for example, in the case:

d is v+v

This is because conventionally we are only interested in the patterns of cause and effect and multiple uses of a value by one definition are not significant.

The directions of the arcs in the graph unambiguously describe cause and effect. If however the graph contains a loop, then the unambiguous nature is lost. We do not allow definition structures with cycles, the simplest of which is a directly self-referential definition, as shown in the upper part of Figure 3.25. Indirect self-reference is also prohibited. An example of indirect self-reference is illustrated in the lower part of Figure 3.25, where *a* is defined in terms of *b* and *b* is defined in terms of *a*.

Detection of graph cycles can be done in two different ways in an implementation. Either graph cycles can be prevented from occurring by making checks at

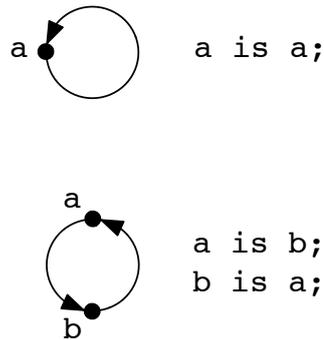


Figure 3.25: Simple graph cycles

redefinition time, or graph cycles can be allowed to occur but detected at evaluation time. The best choice depends, as does the choice of evaluation/storage strategy, upon the balance of redefinition and evaluation occurring in the application. In addition, HODs may complicate detection of graph cycles at redefinition time, since a redefinition to a literal value may cause a HOD-created graph cycle to appear.

Formally, then, the dependency within a definitive script can be described in this way using a *directed acyclic graph*. This term is sometimes shortened to *acyclic digraph* or the even more succinct *dag*.

Graph theory is a large topic which has application to many domains. Consequently there are many textbooks and a large amount of terminology. However, most texts talk of graphs generally and so are of somewhat limited use for information about directed acyclic script graphs. One notable exception is a book by Harary *et al* [HNC65], which is entirely devoted to the subject of directed graphs and has a chapter exclusively about acyclic digraphs. The remainder of this section attempts to give the minimum necessary information useful for an understanding of script graphs.

Harary *et al* [HNC65, p.267] define the ascending level assignment, which is a necessary concept for scheduling the evaluation of nodes and also for drawing a script graph diagram in a particular orientation.

For each point [node] v_i let us denote by n_i the integer assigned to it. A digraph D has an *ascending level assignment*, and the integers are *levels*, if for each line [arc] $v_i v_j$ of D the corresponding integers satisfy $n_i < n_j$. Thus the lines [arcs] of D are directed from lower to higher levels. Of course, if a digraph has a cycle, it has no ascending level assignment.

We shall consider only ascending level assignments and refer to them more briefly as *level assignments* in what follows. In general, there is not a unique level assignment for a given digraph, as the edges in the graph specify only a *partial ordering* on the level assignment. For most purposes we would consider level assignments with fewer levels to be “better” than those with more. The smallest number of levels possible in any level assignment is equal to the number of points on the longest directed path [HNC65, p.270].

The nodes in an acyclic digraph can be ordered by level assignment using an algorithm that performs a topological sort — Knuth’s algorithm mentioned in §3.1.2 is one. However, topological sort algorithms produce only a sequential node ordering, which involves a loss of information relating to nodes that have the same level in a level assignment. On a sequentially evaluating machine this is of little significance, but a concurrent machine can potentially concurrently evaluate nodes at the same level.

Once levels are assigned, it is possible to draw the script digraph in a diagram form where the vertical geometrical relationships of the nodes are meaningful. Conventionally, a *script digraph diagram* is drawn with arcs pointing upwards, and we speak of change propagating *upwards* through the graph and demand-driven evaluation evaluating *downwards*. If the graph is drawn in such a way, then the directions of the arcs need not be shown as it is known that they point upwards.

In 1999, I wrote an Eden script (*vcgWard1999*) to interface the principal EM tool EDEN (see Chapter 4) to a “Visualizer of Compiler Graphs” tool named *xvcg* [San, San96]. Given a list of nodes and edges, *xvcg* can display a graph diagram laid out in a large number of possible ways, including a “fish-eye” layout, which compresses the layout so that a particular node and its immediate surroundings can be seen clearly, but nodes that are further away are reduced in size.

Figure 3.26 shows *xvcg* applied to the Eden *roomviewerYung1991* model. The graph contains 553 nodes and 797 arcs (which are directed in the reverse way to the normal convention). Top-most are the various Eden procedures that propagate internal changes of DoNaLD state to the interface. Near the top is the definition for the screen. At the bottom happens to be the `cart` cartesian coordinate function, which is used by many definitions.

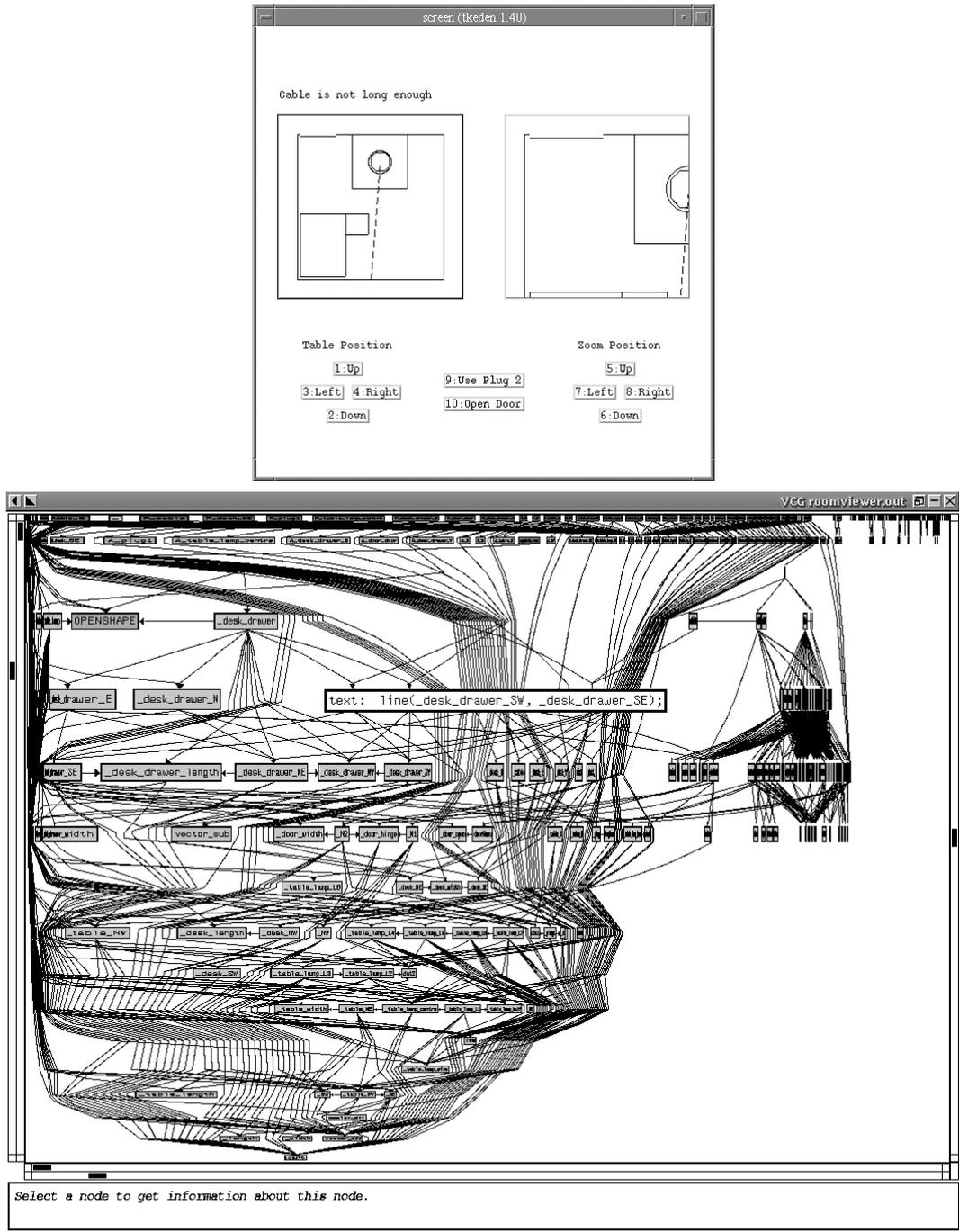


Figure 3.26: xvcg visualisation of *roomviewerYung1991*

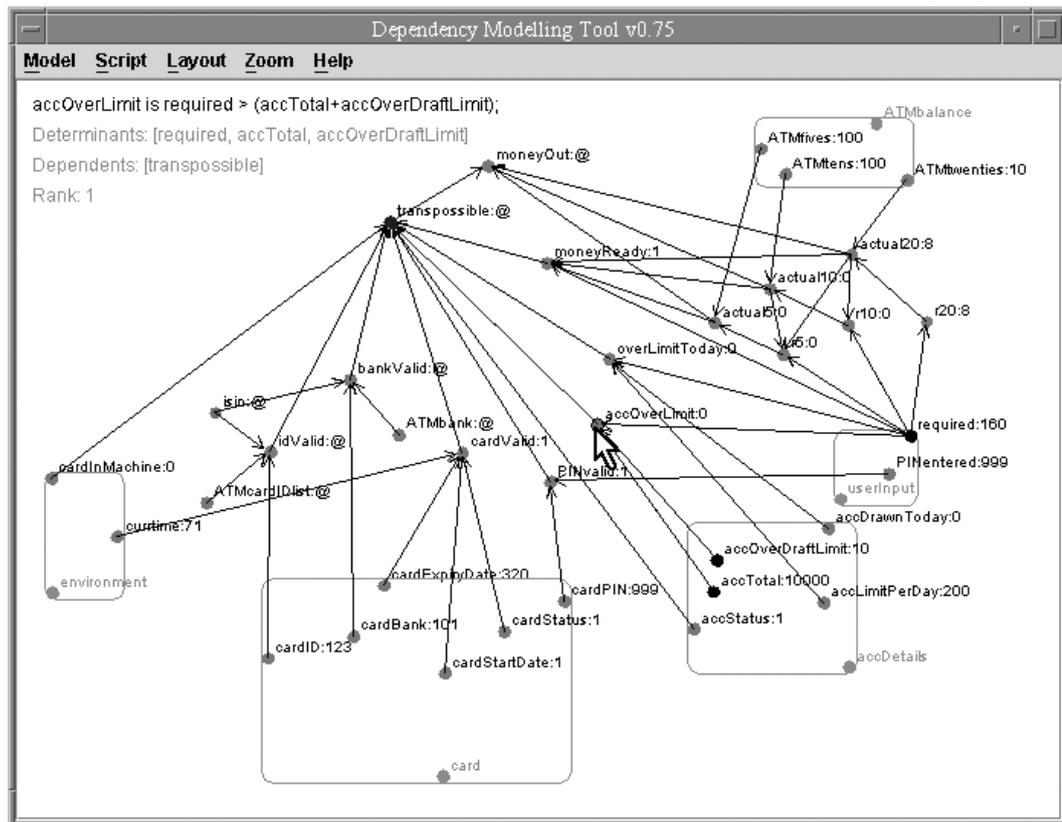


Figure 3.27: Wong’s Dependency Modelling Tool, showing the ATM script

Allan Wong recently implemented a tool named “Dependency Modelling Tool” (DMT) that is based on similar principles — see [Won03, §8]. The DMT reads Eden scripts directly. Three layout algorithms are provided to assist with positioning nodes, which can also be done manually (this is not possible in *xvcg*). The tool implements many forms of interaction with the graph, indicating sources and targets (determinants and dependents in Wong’s terms) of the pointed-to node with colour highlighting. It also allows edges directed toward a node to be hidden and replaced with a round-cornered rectangle in a form of “abstraction”. Figure 3.27 shows the tool applied to a model of an Automated Teller Machine system. Note the “abstractions” and the use of directed edges to denote dependency.

Nodes at the bottom of a script digraph are known as *leaves*. Their values are defined entirely by references to literal values and have no dependencies on other

nodes. Leaves therefore have no incoming arcs:

$$\text{indegree}(\text{leaf}) = 0$$

Leaf nodes can be defined by using only literal values within a definition:

```
a is 42;
```

or through use of an assignment operator, or through the combination of a redefinition with an evaluate-immediately operator (if any of these facilities are provided in the notation used):

```
a := b+c;  
a = | b+c |;
```

Whether these different methods of input give distinct results depends upon the implementation. There is a subtle difference in EDEN, which stores the exact text used on the RHS of a definition, but only the evaluated value from the RHS of an assignment.

Nodes at the top of the digraph are known as *roots* or *sink nodes*. They are not referenced by other nodes. Roots therefore have no outgoing arcs:

$$\text{outdegree}(\text{root}) = 0$$

Contrary to what the name may suggest, there can be more than one root in a script graph. Roots are conventionally drawn at the top of the diagram.

Nodes “inside” the digraph are *internal nodes*. They have both incoming and outgoing arcs:

$$\text{indegree}(\text{internal}) \geq 1 \wedge \text{outdegree}(\text{internal}) \geq 1$$

When a value attached to a node in the script graph or the script graph itself is changed, an internal node can be at the initial or at an intermediate point of a chain of change propagation. If an internal node is itself changed, it will be at the start of the chain. If a node downwards in the script graph is changed, then the internal nodes in the path above the change will be intermediate in the chain, in which case the nodes will be both affected by change and the cause of further change.

Nodes which are not related by dependency to any other node are said to be *isolated* [Wil96, p.12]. An isolated node constitutes a small unconnected subgraph “island” which will not participate in any change propagation:

$$\text{indegree}(\text{isolated}) = \text{outdegree}(\text{isolated}) = 0$$

We often talk of the sources, targets and triggers for a node³⁸.

The *adjacent sources* of a dependency are the nodes to which the dependency directly refers. The *recursive sources* of a dependency are all the nodes in the directed path below the dependency, to which the dependency directly or indirectly refers. The short-hand term *sources* refers to adjacent sources.

The *adjacent targets* of a node are the dependencies that directly refer to the node. The *recursive targets* of a node are all the dependencies in the directed path above the node, which refer directly or indirectly to the node. The short-hand term *targets* refers to adjacent targets.

The set of *triggers* of d is the set of recursive sources of d . A node is said to *trigger* d if it is one of the recursive sources of d .

Two definitive scripts have the same *dependency structure* if their script digraphs are *isomorphic* (simplistically³⁹, if the script digraph diagrams for each could be drawn identically).

The number of possible dependency structures grows large very quickly as the number of nodes increases. Sloane’s On-Line Encyclopedia of Integer Sequences [Slo] has the necessary information as sequence A003087, citing Harary, Palmer and Robinson. Table 3.8 shows the sequence. With even just five nodes, there are 302 possible dependency structures. Beyond five nodes, we quickly obtain extremely large numbers of possibilities.

It is difficult to think about the coordination required in a concurrent definition maintainer without a pool of small example script configurations to draw on. One would sometimes like to find the smallest possible case that exhibits a particular

³⁸Some authors use the terms *dependees* and *dependents/ants*, referring to sources and targets respectively, but these terms are very similar both in text and when spoken — I prefer these terms as laid out originally in [Yun90].

³⁹Formally, two digraphs C and D are isomorphic if D can be obtained from C by re-labelling the nodes — that is, if there is a one-to-one correspondence between the nodes of C and those of D , such that the number of arcs joining any pair of nodes in C is equal to the number of arcs joining the corresponding pair of nodes (in the same direction) in D [WW90, p.82].

N	Acyclic digraphs
0	1
1	1
2	2
3	6
4	31
5	302
6	5984
7	243668
8	20286025
9	3424938010
10	1165948612902
11	797561675349580
12	1094026876269892596
13	3005847365735456265830

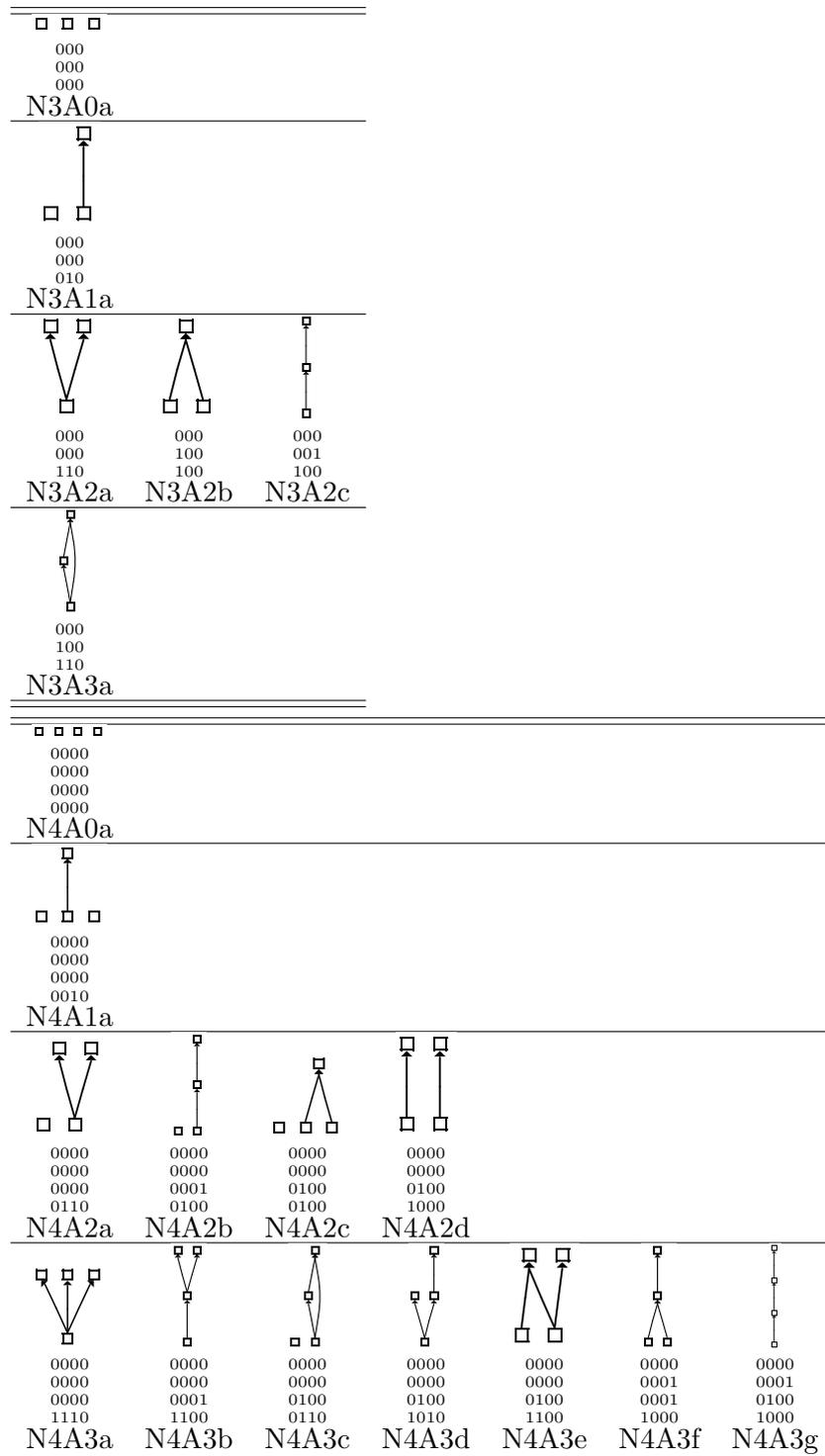
Table 3.8: Number of acyclic digraphs with N unlabelled nodes

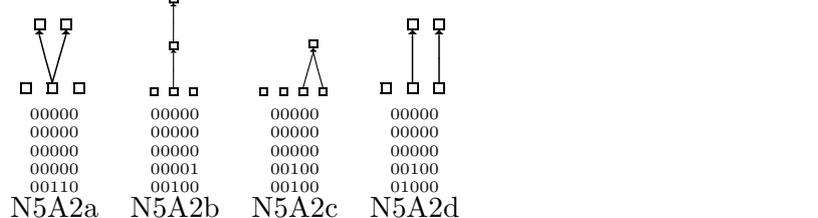
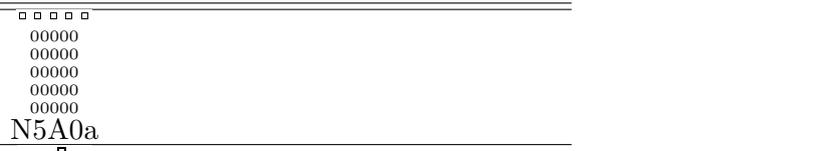
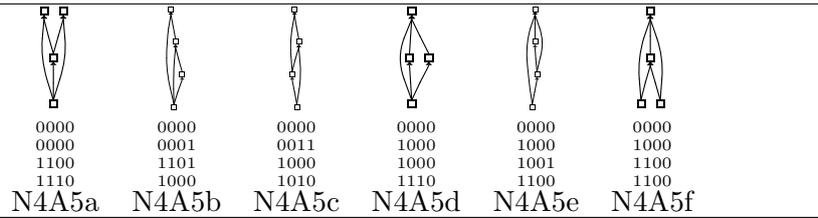
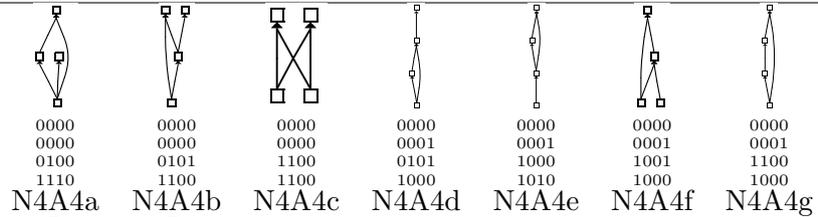
property, for example when constructing the small tests for systematic observation of EDEN presented in §4.3.3. I could not locate a source with a graphical enumeration of acyclic digraphs: [WW90, pp.19–24] contains an enumeration of the 208 unlabelled simple graphs for $N \leq 6$, but Table 3.8 shows there are many more possible acyclic directed graphs; [Car99, Figure 4.8, p.139] purports to show all possible patterns of dependencies for scripts with four definitions, but it omits many possible cases, including all cases with more than three arcs.

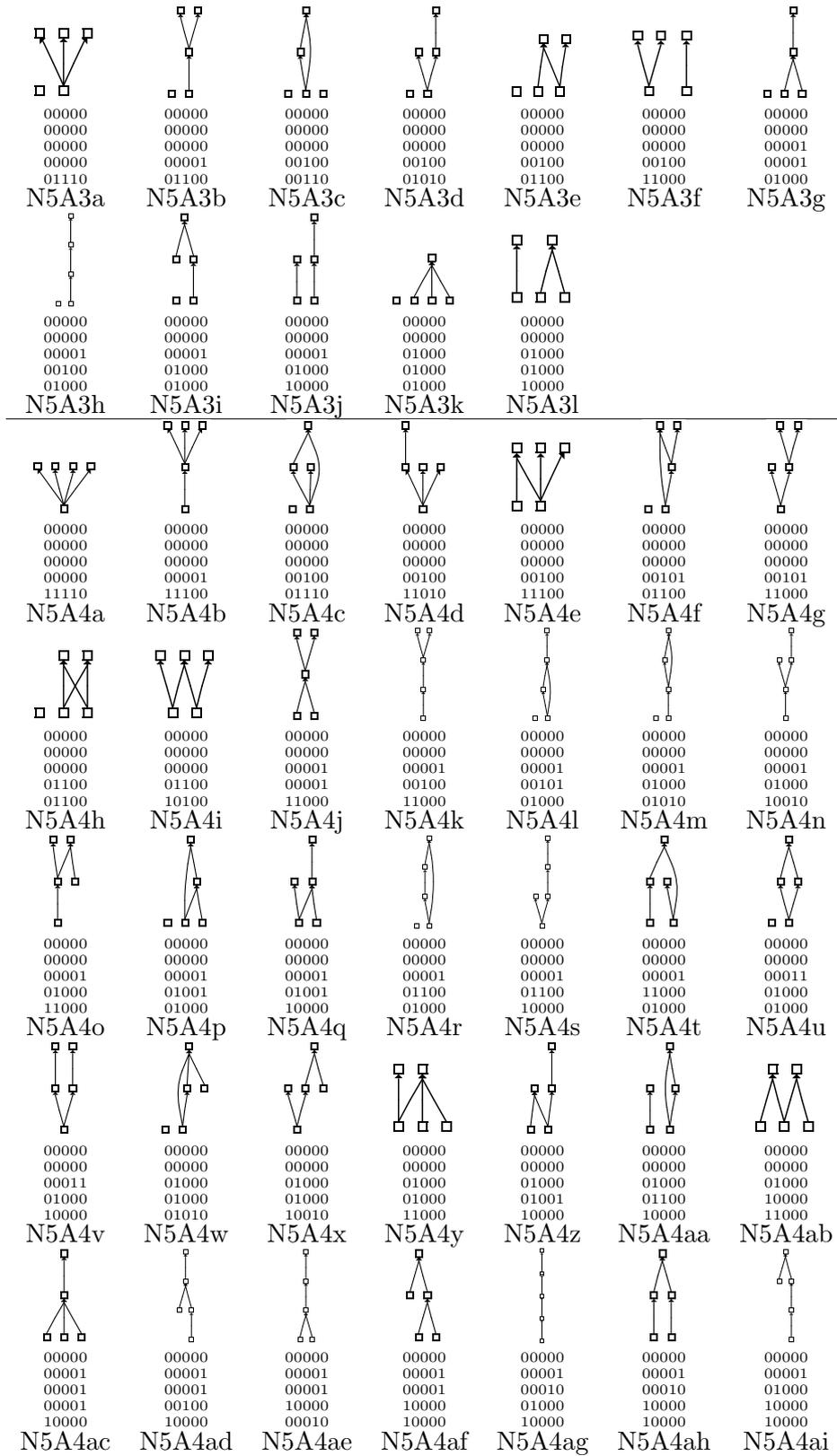
I abandoned a process of obtaining even the results for $N = 4$ by hand, as it was a very error-prone process, and wrote an Eden procedural program to enumerate the acyclic digraphs. The code enumerates all possible $N \times N$ adjacency matrices, treating the values in the adjacency matrix as a 1-dimensional bit vector which is assigned binary values from $00 \cdots 00$ to $11 \cdots 11$ (the bit sequence being $N \times N$ bits long). Each matrix was automatically evaluated to detect cycles and isomorphism with graphs already generated. Cycles were detected using a recursive function to follow arcs, detecting a cycle if a node was visited twice. Isomorphisms were detected by permuting the nodes of the current digraph in all possible ways, using

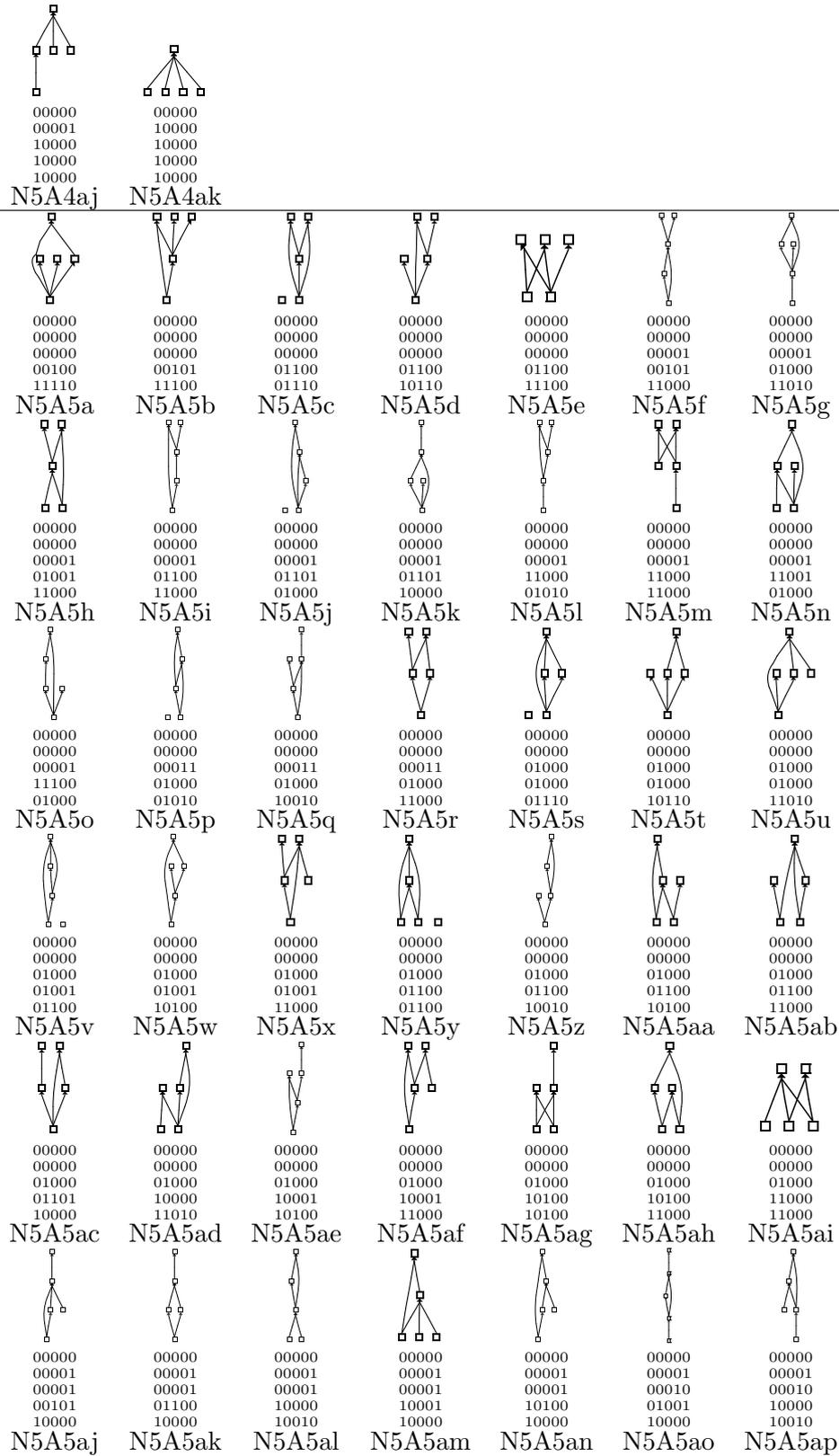
an algorithm based on [Sed90, p.628] and then comparing the permuted versions of the current digraph against the ones already stored. The code does not scale well: it took less than a second to run for $N = 2$ and $N = 3$, about one minute for $N = 4$ and over 14 hours for $N = 5$. Input suitable for the `xvcc` tool was then automatically generated (by Eden code), which was used to lay out and render each graph. Finally the graphs were ordered (again by Eden code) into classes by number of arcs and combined into a table which was rendered using \LaTeX . The results are shown in Figure 3.28.

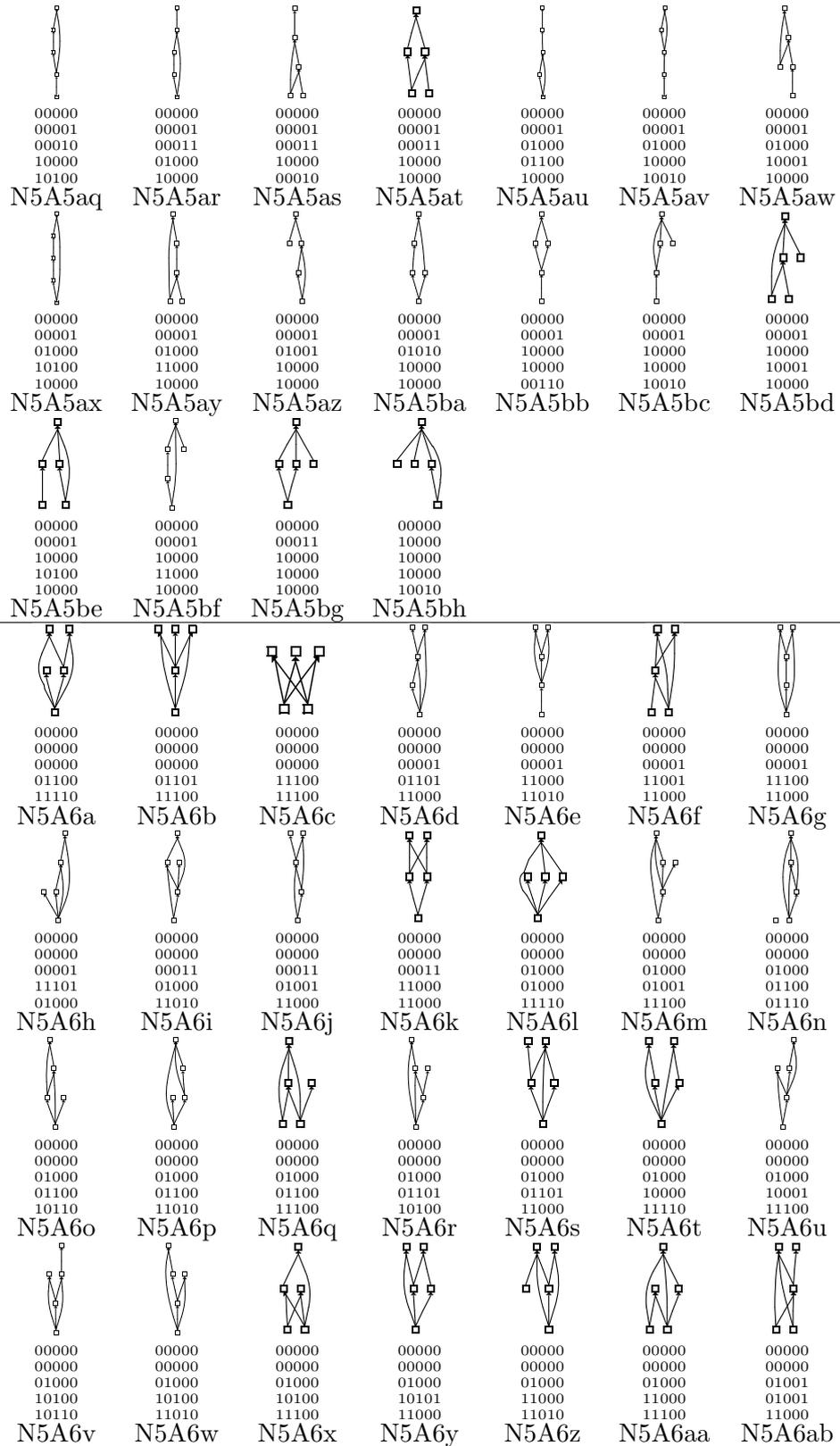
Figure 3.28: Acyclic digraphs graphically enumerated, $3 \leq N \leq 5$

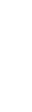
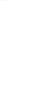
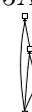
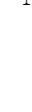
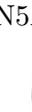
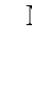
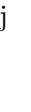


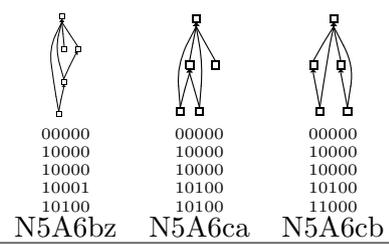








						
00000 00000 01001 01100 11000	00000 00000 01001 10001 11000	00000 00000 01001 10100 11000	00000 00000 01001 11000 11000	00000 00000 00000 11000 11000 11000	00000 00001 00001 01101 10000	00000 00001 00001 10101 10000
N5A6ac	N5A6ad	N5A6ae	N5A6af	N5A6ag	N5A6ah	N5A6ai
						
00000 00001 00001 11100 10000	00000 00001 00010 10000 10110	00000 00001 00011 01001 10000	00000 00001 00011 10000 10010	00000 00001 00011 10001 10000	00000 00001 00011 11000 10000	00000 00001 01000 01101 10000
N5A6aj	N5A6ak	N5A6al	N5A6am	N5A6an	N5A6ao	N5A6ap
						
00000 00001 01000 10101 10000	00000 00001 01000 11001 10000	00000 00001 01000 11100 10000	00000 00001 01001 01001 10000	00000 00001 01001 01100 10000	00000 00001 01001 10000 10010	00000 00001 01001 10001 10000
N5A6aq	N5A6ar	N5A6as	N5A6at	N5A6au	N5A6av	N5A6aw
						
00000 00001 01001 10100 10000	00000 00001 01001 11000 10000	00000 00001 01010 10000 10010	00000 00001 01010 10001 10000	00000 00001 01010 10000 10000	00000 00001 01011 10000 10000	00000 00001 10000 10000 10110
N5A6ax	N5A6ay	N5A6az	N5A6ba	N5A6bb	N5A6bc	N5A6bd
						
00000 00001 10000 10001 10100	00000 00001 10000 10100 00110	00000 00001 10000 10100 10010	00000 00001 10000 10100 10100	00000 00001 10000 10101 10000	00000 00001 10000 11000 10100	00000 00001 10000 11001 10000
N5A6be	N5A6bf	N5A6bg	N5A6bh	N5A6bi	N5A6bj	N5A6bk
						
00000 00001 10000 11100 10000	00000 00001 10001 10000 10000	00000 00001 10001 10100 10000	00000 00001 10001 11000 10000	00000 00001 10010 11000 10000	00000 00001 11000 11000 10000	00000 00011 00011 10000 10000
N5A6bl	N5A6bm	N5A6bn	N5A6bo	N5A6bp	N5A6bq	N5A6br
						
00000 00011 01001 10000 10000	00000 00011 10000 10000 10010	00000 00011 10000 10000 10100	00000 00011 10001 10000 10000	00000 00011 10000 10000 10000	00000 00111 10000 10000 10000	00000 10000 10000 10000 10110
N5A6bs	N5A6bt	N5A6bu	N5A6bv	N5A6bw	N5A6bx	N5A6by



Chapter 4

The Engine for Definitive Notations, EDEN

The EDEN tool is “An Engine for Definitive Notations” [Yun90]. In contrast to `am` and the DAM machine, which use the evaluation/storage strategies 1 (“evaluate-at-use”) and 2 (“evaluate-at-redefinition”) respectively, EDEN uses the hybrid evaluation/storage strategy 3, “evaluate-at-use-when-out-of-date” (see §2.2.1).

This chapter gives an overview of the evolution of the EDEN tool, an evolution which has spanned 17 years at the time of writing. The history is provided in order to place more recent work by the author and others into context. In order to compare EDEN to the ADM and the DAM machine, some of the low-level details of the implementation of the EDEN tool are described and analysed.

The EDEN tool implements the Eden language — in this thesis, I differentiate the two through use of capitalisation. The Eden language (documented in [Yun89]) uses a C-like syntax, providing a conventional procedural scripting language including the procedural assignment operator ‘=’. Unusually, it also provides a way to make (re)definitions, using the definitive ‘is’ construct. The EDEN tool has played a major part in the practice of the EM group and Eden has provided a language for discussion of concepts over many years. The EDEN tool has a variety of front-ends and graphical facilities through the provision of other definitive notations, for example, DoNaLD, a definitive notation for line drawing, and SCOUT, a definitive notation for screen window layout. The various notations are implemented by

translation into the Eden language which is then executed by EDEN.

Eden is thus a procedural-definitive hybrid, allowing both dependency and agency to be programmed or modelled. In this respect, it is more general than the DAM machine, in which the focus is upon dependency. Compared to the ADM, Eden offers a more conventional model. Whilst the ADM could be used in a purely procedural manner by simulating control flow through the use of guards, this would be totally out of keeping with the spirit of intended ADM use.

Automated agency is possible in Eden through the use of a ‘triggered action’ facility. This was initially developed in order to link external graphical state to internal definitive state. As described in §4.3.7, one of the contributions of my work on EDEN has been to discover that the facility can be used to create definitive state definitively.

Automated agency is generally not considered in the DAM machine. Regarding the use of such agency to link external state to internal, consider Allderidge’s attempts to add side effect to operators in order to link external graphics to internal state in §3.3.4. Regarding the use of agency to link state internally, DAM machine operators could invoke `addtoq` in order to modify internal state, but the re-entrant nature of the DAM machine is not well understood (see §3.2.6). The DAM machine also schedules the entire update in phase 2 of the BRA before it is actually performed in step 3 (see Figure 3.3 on p.118). Changes to the script graph as operators are invoked will therefore have no effect on the current `update` process. In contrast, as described in §4.3, EDEN constructs a schedule in a more flexible manner, taking account only of adjacent targets of the ‘current’ node when scheduling change.

Compared to the guarded command lists of the ADM, Eden’s actions are relatively primitive. At some level of abstraction, each Eden action is an ‘atomic’ sequence composed of smaller actions. Eden actions are not interleaved in execution: each runs to completion before the EDEN scheduler determines which to execute next. The invalid transition is not a concept that applies in this sequential scenario, and EDEN makes no checks for conflict between actions. EDEN also lacks two other major features of the (albeit as yet inadequately implemented and little used) ADM:

- entities are not represented — as a result, instantiation and removal of blocks of definitions involves parsing the symbol table as a set of strings, and
- there is no manual control ‘debugger-like’ stepping mode.

This chapter is organised as three main sections. Section §4.1 discusses the development of EDEN from its first conception by Y.W. Yung and the further developments until 1999 by Y.P. Yung (younger brother of Y.W. Yung) and P-H. Sun. Section §4.2 describes the principal highlights of my contribution to the development of EDEN since 1999. Section §4.3 gives an analysis of the operational semantics of EDEN.

4.1 EDEN, chronologically to 1999

4.1.1 The early history

The EDEN tool has received comparatively more attention than the ADM and the DAM machine, and the number of written sources is correspondingly higher. Focussing on the machine rather than applications, however, leads to four primary sources [Yun87, Yun90, Yun93, Yun96] from the two original authors of EDEN which form the basis for much of the analysis in this chapter. This section presents the history of the Eden language and the various implementations. It does not discuss the operation of the definition maintainer, which is discussed in §4.3.

Y.W. ‘Edward’ Yung was the original author of the Eden language and the first terminal-based implementation which we now call `ttyeden`¹. The `ttyeden` implementation and hence some parts of the language are based on a tutorial example program included in [KP84], named `hoc`. The explanations that follow are a retrospective rewrite of EDEN history: Y.W. Yung’s writings justify and explain EDEN largely independently of `hoc`, but the perspective taken below is useful to highlight the distinction between “standard textbook implementation” and novel definitive features.

Hoc (“high-order calculator”) is “an interactive language for floating point arithmetic”, written in the C language with the assistance of the `yacc` parser generator

¹We will use this name throughout to identify the terminal-based implementation of the Eden language, even though the name was not used at this early time.

and the `lex` lexical analyzer. It is similar to the standard UNIX arbitrary-precision calculator utility `bc`. The implementation presented in [KP84] provides a four-function calculator with constants such as `PI`; built-in functions such as `sin`; an assignment statement acting on procedural variables; relational operators and control flow; recursive procedures and functions with arguments, and input/output routines. The following example is combined from several [KP84, p.234, p.245, p.242, p.246, p.274, p.332, p.331] given in the book for use of `hoc` and gives a fairly complete idea of the language facilities available. Keyboard input is shown `like this`.

```

$ hoc
4*3*2
    24
(1+2) * (3+4)
    21
355/113
    3.1415929
x = 355
y = 113
x/y
    3.1415929
x = y = z = 0
sin(PI/2)
    1
proc fib() {
    a = 0
    b = 1
    while (b < $1) {
        print b
        c = b
        b = a+b
        a = c
    }
    print "\n"
}
fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
func fac() {
    if ($1 <= 0) return 1 else return $1 * fac($1-1)
}
fac(7)

```

```

5040
func ack() {
  if ($1 == 0) return $2+1
  if ($2 == 0) return ack($1-1, 1)
  return ack($1-1, ack($1, $2-1))
}
ack(3, 2)
29
while (read(x)) {
  print "value is ", x, "\n"
}
42
value is 42

```

Listing 4.1: An interaction with hoc

There are significant differences between `hoc` and `ttyeden`. These include:

- The data types supported are very different from `hoc`. The early version of EDEN had no floating point type. Integer, character, string and pointer data types, the special undefined type ('@') and a heterogeneous list type was included to allow the preceding atomic types to be composed into new data types.
- Some further built-in functions were added to `ttyeden`, mainly to manipulate the new data types.
- An interface was developed to the C library, allowing the addition of functions such as `fprintf`. In particular, 41 functions from the 'curses' terminal window handling package were added.
- Various procedural statements were added to extend the available control flow options (`break`; `continue`; `do`; `for`; `switch`; `case`; `default`), implement variables local to a `proc` or `func` (`auto`) and to manipulate the list data type (`shift`; `append`; `insert`; `delete`).

The introduction of the above features enhances the computational power of the resulting tool, but does not change the programming paradigm. From the perspective of this thesis, other extensions of `hoc` entailed in developing `ttyeden` are far

more important. These include mechanisms for managing dependency that have been so seminal in the development of EM that they merit a separate subsection.

4.1.2 Formula variables and actions

The more radical and significant features that were added to `hoc` in developing `ttyeden` are listed below.

- Formula variables;
- Actions (automatically invoked procedures);
- An ‘auto-recalculate’ mechanism.

The auto-recalculate mechanism is the subject of §4.3 below. At this point, it will suffice to say that the auto-recalculate mechanism implements automatic evaluation of formulae and actions at the correct time. The automatic evaluation can be temporarily delayed by setting the Eden `autocalc` variable to false.

The variables that are present in `hoc` are known in Eden as Read-Write Variables (RWVs²). RWVs are assigned to using the `=` operator, using the conventional syntax *identifier = expression*. For example:

```
a = a+1;
```

Formula variables (FVs) in Eden are distinct from RWVs. Y.W. Yung has the following to say [Yun90, p.27] about the distinction between FVs and RWVs.

EDEN supports the concept of formula. A formula definition has the form:

identifier is expression ;

or

F is $\phi(v_1, v_2, \dots, v_n)$;

The keyword ‘`is`’ defines a formula variable F whose value is computed from the values of source variables v_1, v_2, \dots, v_n and the expression on the right hand side, denoted by ϕ , is the formula of the variable F . In other words, a formula describes how the value of a variable is computed from other data. These formulae are permanently valid (unless they are redefined). That is, no matter what the values of source variables are, the value of variable F is *always* equal to $\phi(v_1, v_2, \dots, v_n)$. Thus a formula gives an abstract definition

²The term RWV is introduced in [Yun90] — [Yun87] uses the term “value variable”.

of a variable rather than the explicit value of it. This is the major difference between formula definitions and conventional assignment statements (denoted by the = operator). For example, after executing the assignment,

$$V = \phi(v_1, v_2, \dots, v_n);$$

the value of V is equal to $\phi(v_1, v_2, \dots, v_n)$ only after the expression ϕ is evaluated and before any of the values of the source variables is altered. In other words, the assignment operator takes a snapshot of the value of ϕ at the instant of evaluation and stores the result in the variable V .

Actions in Eden build on the procedure- and function-call subroutine mechanism from hoc. They are procedures which are automatically invoked when a named FV or RWV changes value. Y.W. Yung [Yun90, p.30] outlines the syntax and motivation for actions as follows.

The general form of an action specification in EDEN is:

```
proc identifier : identifier-list { C-like-statements }
```

The following statement illustrates a sample action definition.

```
proc display_v : v { writeln(v); }
```

The keyword `proc` defines an action, named as `display_v`, which is invoked by the system when the value of variable `v` (specified after the colon) is changed (the meaning of ‘*changed*’ shall be discussed later). The curly brackets `{ }` enclose a list of statements to be executed sequentially. In this case, there is only one function call, the `writeln` function. By calling different functions (with appropriate side-effects, e.g. `writeln(...)` prints the values of its arguments on the standard output) in the function libraries, `display_v` can do different tasks, such as realize the data graphically. This makes EDEN more flexible.

Later, Y.W. Yung [Yun90, p.37] defines the intent of the word *changed*.

An action specification (AS) is a named sequence of instructions. This sequence of instructions will be invoked by the system whenever the values of any source variables, specified explicitly in a list, are *changed*. The term ‘*changed*’ is causally defined. It may mean the value of a variable is different from the previous one, or the value of a variable is overwritten (by the user or by the system) though the value may be the same as the previous value. EDEN takes the latter definition.

An alternative way of defining actions which “is sometimes more appropriate” is described in [Yun87, p.13]. The syntax is:

```
variable ~> [ procedure_list ] ;
```

The example shown above can be written in this alternate form as:

```
proc display_v { writeln(v); }  
v ~> [display_v];
```

```

xterm
.....1.....2.....3.....4.....5.....6
Notes on using Edward Yung's text editor
(by Ashley Ward, May 1999)

ttyeden editor.e
Provide a filename - text

vi style keys:
^H left
^J/^V down
^K up
^L right

^E end of line
^N bottom of window
^O top of window

^T toggle insert mode
.....1.....2.....3.....4.....5.....6
line=1                               insert
top of file

```

Figure 4.1: *texteditorYung1987* running in *ttyeden-1.52*

Y.W. Yung [Yun87, p.13] suggests that the syntax ‘v ~> [display_v]’ can be read as “as v is changed, do display_v”.

4.1.3 *texteditorYung1987*

The first non-trivial definitive model to be written in Eden was the major example given in [Yun87], which I have filed in the empublic archive [WRB] as *texteditorYung1987*. The model implements a terminal-based ‘visual’ text editor, rather like the standard editor *vi*. The model still runs in the current version of *ttyeden* that I maintain 17 years later, in 2004: *ttyeden-1.52*. Figure 4.1 shows the text editor being used to edit some text that describes how to use it.

In this section, I present an analytical review of the *texteditorYung1987* model in order to demonstrate the principal features of *ttyeden* (the ‘purest’ Eden implementation) which were briefly outlined in the previous two subsections. It is also interesting to compare this model to the example given in §3.5.4 where the DAM machine is configured to display three character patterns on the screen, the pixels making up the glyphs being dependent on the text data.

Firstly, I show parts³ of the model in order to demonstrate use of RWVs, FVs,

³I only give extracts as the original is some 479 lines long.

```

1 text = [""];
2 line = 1;
3 col = 1;
4 insert_mode = 1;    /* default insert_mode = ON */
5
6 WIN_WIDTH = 60;
7 WIN_HEIGHT = 20;
8 win_left = 1;
9
10 RULER = repeatchar(MAXCOL, '.');
11 for(i = 5 ; i <= MAXCOL; i += 10) RULER[i] = ':';
12 for(i = 10; i <= MAXCOL; i += 10) RULER[i] = char(i/10 % 10 + '0');

```

Listing 4.2: A selection of Read-Write Variables from *texteditorYung1987*

actions, funcs and procs in Eden. The order of investigation represents one way in which an analysis of a definitive model can usefully proceed, unusually starting with the data rather than the actions.

In the final part of this section, I show the powerful potential for the use of dependency, even in this terminal-only variant of EDEN, by introducing some re-definitions to *texteditorYung1987*.

RWVs (Read-Write Variables)

Listing 4.2 shows a selection of Read-Write Variables that define the text that is being edited, which is initially blank (line 1); the cursor position within the text (lines 2 and 3); and whether the editor is in ‘replace’ (‘over-type’) or ‘insert’ mode (line 4). Next, the size of the terminal window is assigned (lines 6 and 7). The terminal window will sometimes be narrower than the full text, and the editor is able to scroll the window to display just a portion of the text. The RWV `win_left` (line 8) holds the index of the leftmost displayed character.

Figure 4.1 shows a ‘ruler’ displayed at the top and bottom of the text, which gives an indication of the column indices. The `RULER` RWV string that will be shown as the ruler is initialised by first using a function to create a line of periods (line 10), some of which are then overwritten by the two `for` loops that follow (lines 11 and 12), which introduce the positional markings into the ruler string.

```

1 CUR_LINE is (line > text#) ? "" : text[line];
2 CUR_CHAR is (col > CUR_LINE#) ? ' ' : CUR_LINE[col];
3
4 win_right is win_left + WIN_WIDTH - 1;
5
6 CUR_RULER is substr(RULER, win_left, win_right);
7
8 MAXCOL is 100;

```

Listing 4.3: Some Formula Variables in *texteditorYung1987***FVs (Formula Variables)**

Some examples of Formula Variables are shown in Listing 4.3. FVs define a string containing the current line of text (line 1) and current character (line 2) at the cursor position, with blank substitutions if the cursor has moved beyond the limits of the current text⁴. The index of the rightmost displayed character, `win_right` (line 4), is calculated from the `win_left` and `WIN_WIDTH` RWVs shown in Listing 4.2. The portion of the ruler string which is currently on the screen (`CUR_RULER`, line 6) is calculated by finding the appropriate substring of `RULER`. The last FV here (`MAXCOL`, line 8) is given as an example to show that FVs can simply be defined to literal values, in which case they act like a RWV.

Actions

The text editor model contains many actions, most of which eventually call the ‘curses’ C library routines to change the terminal window state. Some examples are shown in Listing 4.4. The `~>` syntax is used to cause a change of variable to invoke a procedure call. The listing specifies how a change to the `message` RWV causes (line 18) a call to the `print_message` procedure action (line 1) and then the `update_scrn` action (line 13), which together cause the value of the `message` variable to appear on the bottom line of the terminal. (See the indication “top of file” shown in Figure 4.1 for example.) Any change in the `insert_mode` of the editor, or to the displayed portion of the ruler (`CUR_RULER`), is propagated to the display in a similar way (lines 19,20).

⁴The `#` operator gives the number of items in a list/characters in a string, and `?:` is the ternary ‘if-then-else’ operator, as provided in the language C.

```

1 proc print_message {
2     if (message == @) return;
3     move(WIN_HEIGHT + 3, 0);
4     addstr(message);
5     addch('\n');
6 }
7
8 proc print_insert_mode {
9     move(WIN_HEIGHT + 2, 33);
10    addstr(insert_mode ? " insert" : "replace" );
11 }
12
13 proc update_scrn {
14     at(line, col);
15     refresh();
16 }
17
18 message ~> [print_message,          update_scrn];
19 insert_mode ~> [print_insert_mode,    update_scrn];
20 CUR_RULER ~> [print_ruler, renew_scrn, update_scrn];

```

Listing 4.4: Some actions in *texteditorYung1987*

Functions

Functions in Eden are very similar to those in hoc. The text editor model defines a few utility functions, including the following function that returns the maximum of two provided parameters:

```
func max { return $1 > $2 ? $1 : $2 ; }
```

Procedures

Finally, procedures are used to encapsulate and name state change. Some procedures from the *texteditorYung1987* model are shown in Listing 4.5. The procedure `right` is called to move the cursor one place to the right. If the cursor position exceeds the maximum line length, then the `message` RWV is set to show an appropriate error (line 5) and is then automatically displayed on the terminal by the `print_message` and `update_scrn` actions shown in Listing 4.4.

```

1  proc right {
2      if (col < MAXCOL)
3          col++;
4      else
5          message = "right margin";
6  }
7
8  proc replace_char {
9      if (col <= CUR_LINE#) {
10         text[line][col] = $1;
11         addch($1);
12         right();
13     } else
14         insert_char($1);
15 }

```

Listing 4.5: Some procedures in *texteditorYung1987*

The procedure `replace_char` (line 8) is called when a key is pressed and the editor is in ‘replace’ (rather than ‘insert’) mode. The procedure modifies the `text` RWV and calls the curses `addch` routine to make the corresponding change on the terminal. Notice that changes to the `text` RWV are propagated to the terminal in this *ad hoc* manner rather than by using dependency in the manner of the dependent pixels example in §3.5.4. This is probably due to problems with using lists in dependencies, which we shall describe in §5.2.1. The *ad hoc* propagation of change can cause problems in synchronising the EDEN state and the terminal state, as we shall see shortly.

The `edit` procedure is invoked to start the editor. A selection of lines from this moderately long procedure are shown in Listing 4.6. The procedure repeatedly waits to get a key press (the `fgetc` call on line 4) and then calls the appropriate procedure to process it. For example, the `right()` procedure is called (on line 17) when control-L (the key press for ‘move cursor right’ in the `vi` editor) is pressed.

Whilst the `edit` procedure is blocked waiting for a key press, the entire `ttyeden` process is blocked, as EDEN is a sequential, single-threaded machine. The `edit` procedure therefore has exclusive control over EDEN when it is running. The control-D key command terminates the `edit` procedure (line 14) and `ttyeden` can then be used interactively in the normal way (using the Eden language) again.

```
1 proc edit {
2     auto c;
3     ...
4     while (mycbreak() && (c = fgetc(stdin)) != EOF) {
5         if (message != "")
6             message = "";
7         if (c < ' ') {
8             switch (c + 'A' - 1) {
9             ...
10            case 'D':      /* ^D */
11                reset();
12                move(WIN_HEIGHT+3, 0);
13                refresh();
14                return;
15            ...
16            case 'L':     /* ^L */
17                right();
18                break;
19            ...
20            case 'T':     /* ^T */
21                insert_mode = ! insert_mode;
22                break;
23            ...
24            } else if (c == 127) { /* DEL */
25                delete_char();
26            } else if (isprint(c)) {
27                c = char(c);
28                if (insert_mode)
29                    insert_char(c);
30                else
31                    replace_char(c);
32            }
33        }
34    }
```

Listing 4.6: Highlights from the *texteditorYung1987* edit procedure

```

1  RULERPROMPT is str(text#) // " lines in total";
2  CUR_RULER is substr(RULER, win_left, win_left+5) // RULERPROMPT //
3     substr(RULER, win_left+5+RULERPROMPT#, win_right);
4
5  WIN_WIDTH is col + 10;
6
7  insert_mode is CUR_CHAR != 'a';
8
9  func st {
10     auto s, ret;
11     ret = [];
12     s = symbols("formula") // symbols("var") // symbols("proc") //
13         symbols("func");
14     while (s != []) {
15         if (symboldetail(s[1])[6] != "system")
16             append ret, symboldefinition(s[1]);
17         shift s;
18     }
19     return ret;
20 }
21 text is st();

```

Listing 4.7: Some possible redefinitions to make to *texteditorYung1987*

Redefinitions

The model does not use a large number of FVs. In this final part of the section, I show how judicious introduction of more FVs to the model can make interesting modifications to the text editor, beyond what Y.W. Yung would have conceived. For example, I introduce FVs where — expecting the data to be exclusively under program control — he used RWVs. As well as demonstrating the potential for change that is not preconceived, the redefinition examples also serve to illustrate various limitations of our current implementation of dependency.

Four examples of the use of dependency are given in Listing 4.7. They perform the following modifications to the text editor:

- Modify the ruler to indicate the total number of lines in the text;
- Make the terminal window width dependent upon the cursor position;
- Make the editor insert/replace mode dependent upon the current cursor character;
- Make the editor show its own Eden code.

The first pair of definitions (lines 1–3 of Listing 4.7) change the ruler to include information about the total number of lines in the edited text. The `//` operator is used to concatenate strings in Eden. Two definitions are used to cope with the fact that the number of lines in decimal is a variable number of characters long, yet `CUR_RULER` needs to be exactly `win_right - win_left` characters long. After these definitions are introduced, the ruler is automatically updated with the new information when the text is modified. Note that the `RULERPROMPT` variable (and hence the `CUR_RULER` variable) need only be updated when the number of items in the Eden `text` list variable changes, but it is actually updated whenever the `text` variable changes in any way.

The ruler was defined as a FV in the original model. RWVs can also be changed to FVs with unusual but perhaps useful results. For example, the length of the lines in the display can be defined to be always ten more than the current horizontal cursor position (line 5). Moving the cursor to the right then causes the display to ‘expand’. Above, we mentioned the *ad hoc* way in which changes are propagated to the terminal by the model: the `replace_char` procedure shown in Listing 4.5 calls the curses `addch` routine as a side-effect. The internal state of the `text` variable is not linked to the external state on the terminal by dependency. As a result, moving the cursor to the left does not immediately contract the terminal window, as there is no action in the model to output clearing blanks to the right of the current display.

The editor can be made to automatically change to ‘replace’ mode when the cursor is over an ‘a’ character and insert mode otherwise, with the redefinition shown on line 7 of Listing 4.7. This example is given partially to illustrate a problem, as follows. If `insert_mode` is defined as a FV, as shown on line 7, and then the user uses the “toggle insert mode” key press (control-T), the `edit` procedure will make a procedural assignment (using `=`) to the `insert_mode` variable (on line 21 of Listing 4.6), changing it back from a FV to a RWV. This illustrates that dependencies can be easily destroyed with a careless procedural assignment in EDEN.

Finally, the function and definition shown on lines 9–21 of Listing 4.7 cause the text editor to show its own Eden code⁵. Further developments involving “in itself” aspects of Eden are described in §4.2. This particular example is given here in order

⁵The `st` function shown requires a facility for identifying ‘system’-defined variables which is available in version 1.52.

```

xterm
.....:67 lines in total...:.....3.....:.....
CUR_CHAR is (col > CUR_LINE#) ? " " : C
win_offset_y is 1;
win_offset_x is 0;
MAXCOL is 100;
CUR_LINE is (line > text#) ? "" : text[
text is st();
win_bottom is win_top + WIN_HEIGHT - 1;
EndOfFile is "end of file";
TopOfFile is "top of file";
CUR_RULER is substr(RULER, win_left, wi
WIN_WIDTH is col + 10;
insert_mode is CUR_CHAR != " ";
win_right is win_left + WIN_WIDTH - 1;
RULERPROMPT is str(text#) // " lines in
filename="using.txt";
autocalc=1;
line=1;
WIN_HEIGHT=20;
win_left=1;
win_top=1;
.....:67 lines in total...:.....3.....:.....
line=12
replace

```

Figure 4.2: *texteditorYung1987* with some example redefinitions

to illustrate two issues. Firstly, note that although this example successfully displays the model state in the text editor, the text cannot be edited, as the dependency `text is st()` implies only a one-way constraint. Secondly, note that if the text editor model state changes (when a redefinition is made for example), the text editor does not automatically re-evaluate the `st()` function and display the new state. There is no way to specify the necessary dependency in the current Eden.

The modified text editor with these four sets of redefinitions is shown in Figure 4.2.

4.1.4 DoNaLD, SCOUT and ARCA pipeline translators

Although Eden is a powerful general-purpose definitive language, we desire domain-specific definitive notations to assist with our modelling. ‘Pure’ definitive notations comprise only (re)definitions, of the form $id = expr$. They are based purely on dependency. Agency is limited to that of the modeller, so a pure definitive notation provides a 1-agent modelling environment.

The first definitive notation to be implemented with EDEN was DoNaLD, a Definitive Notation for Line Drawing. DoNaLD was specified before the advent of EDEN, in [BABH86]. Y.W. Yung was the author of the original implementation,

and describes it in [Yun90, §6]. Y.P. Yung designed and implemented the SCOUT definitive notation (which describes S**Screen lay**OUT) using the same strategy for implementation. The strategy for implementing a definitive notation in EDEN, codified by Y.P. Yung in [Yun93, §6.3.1], is quoted below as it provides the explanation behind the resulting translator. It is also a nice example of incremental construction and “human computing”, where a process is first attempted manually and later automated.

1. Derive a scheme for translating [definitive notation] variable names into Eden variable names. For example:

DoNaLD name	Eden name
<code>table</code>	<code>_table</code>
<code>table/drawer</code>	<code>_table_drawer</code>
<code>table/drawer/width</code>	<code>_table_drawer_width</code>

2. Emulate the data types and operators using Eden data types and user-defined functions. Almost inevitably this will make use of the list structure in Eden because list is the only complex data type in Eden. For example:

DoNaLD type	Eden type
<code>integer</code>	<code>integer</code>
<code>point</code>	<code>['C', integer, integer]</code>
<code>line</code>	<code>['L', point, point]</code>

DoNaLD operator	Eden operator/function
<code>div</code>	<code>/</code>
<code>+ (vector sum)</code>	<pre>func vector_add { para p1, p2; return ['C', p1[2]+p2[2], p1[3]+p2[3]]; }</pre>

3. The underlying algebra of the target notation has been implemented through steps 1 and 2. To complete the implementation, the required implicit actions are emulated using Eden’s user-defined actions. For example:

DoNaLD code	Eden action specification
<code>integer i</code>	No action
<code>point p</code>	<code>proc P_p: _p { plot_point(&p); }</code>
<code>line L</code>	<code>proc P_L: _L { plot_line(&L); }</code>

Notes: No action is required for integer `i` because integer variables do not have any graphical representation in DoNaLD. The `&` operator is similar to that in the C language: it returns the address of the variable. `plot_point` and `plot_line` are Eden (user-defined) procedures which do the plotting.

4. Write a preprocessor to translate scripts in the definitive notation into Eden in the way implicitly defined by steps 1 to 3.

The original DoNaLD implementation communicated with EDEN through a UNIX pipeline. Y.W. Yung [Yun90, p.89] gives the details.

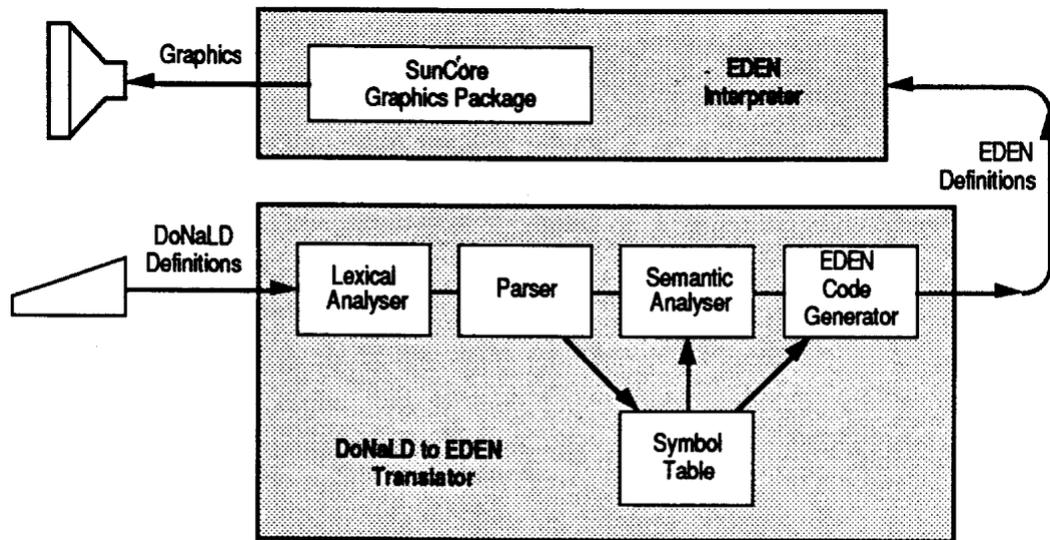


Figure 4.3: The original DoNaLD system (from [Yun90, Figure 6-1])

The following UNIX shell script, `Donald`, forms the complete DoNaLD system:

```
donald.translator | eden -n init.e
```

where `donald.translator` and `eden` are the DoNaLD to Eden translator and EDEN interpreter respectively.

The `donald.translator` is a filter where DoNaLD specification is read from the standard input file and the EDEN code is output to the standard output file (i.e. pass to `eden`).

The `-n` option tells `eden` to run in non-interactive mode (i.e. no prompt). The file `init.e` stores all supporting EDEN codes which is loaded before `eden` reads the standard input (i.e. the output of `donald.translator`).

The `donald.translator` program was constructed using the conventional `lex` and `yacc` tools. The actual graphics were generated by EDEN procedures which called the SunCore C library package available at the time. The whole system is illustrated in Figure 4.3.

Much work on definitive notation translators was performed at this time: the DoNaLD translator was later enhanced by Chan [Cha89] and Parsons [Par91]. The original SCOUT and ADM (see §2.2.2) translators were implemented by Y.P. Yung in a similar manner to DoNaLD. Stuart Bird wrote the first ARCA⁶ to Eden

⁶A notation for describing Cayley diagrams.

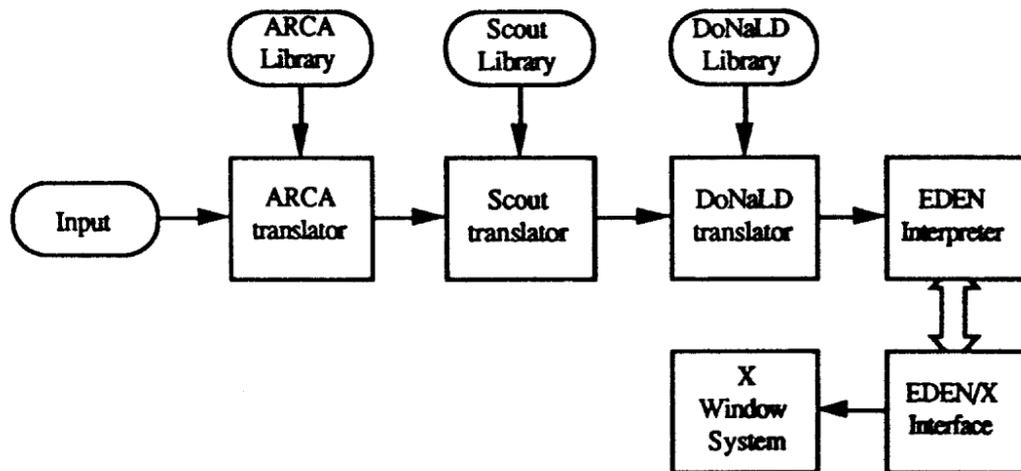


Figure 4.4: ARCA, SCOUT and DoNaLD combined on a pipeline (from [Yun93, Figure 6-5])

translator *arcaBird1991*, again implemented in a similar manner, which was later modified by Y.P. Yung and then myself (*arcaWard2002*).

By 1992, it was possible to integrate all these definitive notations using UNIX pipelines, and an ‘EDEN/X’ (EX) interface had been constructed to display results using the new X Window client-server graphics system. Y.P. Yung named the result “the SCOUT system”. The pipeline is illustrated in Figure 4.4 and described in the quote below from [Yun93, p.95].

The SCOUT system is implemented in a UNIX environment. The user’s input is pipelined through a series of ARCA, SCOUT and DoNaLD filters (the ordering of the filters is not important) which translate ARCA, SCOUT and DoNaLD definitions into Eden definitions. These definitions together with the functions and actions in the libraries are interpreted by the EDEN interpreter. Graphical outputs are generated by an EDEN/X interface which is actually an X Window client. When the graphics display needs to be updated (i.e. some EDEN actions generate graphics output), the EDEN interpreter will interact with the EDEN/X interface, which in turn will interact with the X Window server to produce graphics images.

Note that there is only one input, and so a mechanism must be devised to control which translators in the pipeline will act on the input. Y.P. Yung [Yun93, p.99] describes the details in the quote below.

In order to incorporate several definitive notations into a single system, the SCOUT system uses a method similar to the way the preprocessors of `roff` (the standard UNIX text formatting language) works. A block of definitions of a definitive notation is preceded by a declaration of the notation name. For example:

```
%scout
...
SCOUT definitions
...
%donald
...
DoNaLD definitions
...
```

The individual translator will translate only the lines from the notation declaration downwards until the declaration of another definitive notation is reached; the remaining lines are untouched. In this method, all the translators are running independently of one another.

Many different ways of organising the communication between individual translators of definitive notations are represented in the versions of EDEN developed by Y.W. Yung and Y.P. Yung between 1989 and 1996. Several key issues⁷ are raised. These relate to the extent to which EDEN:

- can itself generate input for translators;
- can hold an image of the overall system state for use in the interface to the modeller;
- supports interleaving of actions on the part of automated and human agents, and
- permits the experimental development of new notations.

The next section discusses the role played by some of these issues in Y.P. Yung's development of `tkeden`. Section §4.2.4 describes a generalised notations framework that I have developed that has the potential to address all the above issues.

⁷At this time, efficiency was also a significant concern, but this is no longer relevant largely, due to "Moore's law" (see §4.1.7) and the fact that typical Eden models are small as the bulk of the definitions in a script are usually hand-crafted.

4.1.5 The early *tkeden*

In developing *tkeden*, Y.P. ‘Simon’ Yung set out to integrate several definitive notations into one tool, using the then-new Tcl/Tk graphics system [Ous94]. He explains the motivation and results in [Yun96, p.7].

The shortcoming of [the SCOUT] system is that interactions between different modules are restricted. The pipeline cannot be wrapped round. Therefore, EDEN cannot easily and efficiently generate DoNaLD or SCOUT definitions and pass them to the appropriate translator located at the front of the pipeline. Also, the separation of the translators, the EDEN interpreter and the graphical interface means that information about the overall system state cannot easily be examined.

In order to solve these problems, the integrated environment *xeden* and its successor *tkeden* are developed. *Xeden* combines the DoNaLD translator, the SCOUT translator, the EDEN interpreter as well as EX into one unit. Apart from the speed improvement, *xeden* can then use the existing EDEN `execute()` command to evaluate strings representing a DoNaLD or SCOUT script. The DoNaLD `graph` function⁸ is later implemented based on this newly established capability.

tkeden is developed from *xeden*. Instead of using the core of our home-grown program EX as the interface to X, we make use of Tk as the graphics and event driver. The advantages of using Tk include:

1. faster graphics (compared to EX),
2. 3D look graphical interface: SCOUT windows are now 3D,
3. easier to build multi-window graphical user interface,
4. easier to extend our graphics notations because Tcl/Tk is a script language,
5. better connectivity to other programs because Tk has a builtin inter-process communication mechanism,
6. possibility of porting to non-X environment due to the portability of Tcl/Tk.

... *tkeden*, over and above *xeden*, provides the following features:

- different views of the definition stores;
- save and load the current set of definitions;
- view and save the history of interaction.

The tool that Y.P. Yung produced, *tkeden*, included the DoNaLD and SCOUT notations. Figure 4.5 shows an early version of *tkeden* running *cruisecontrol-Bridge1991*, which uses Eden, DoNaLD and SCOUT to model a vehicle with a cruise control facility progressing up and down a sinusoidal hill.

⁸A higher-order definition feature.

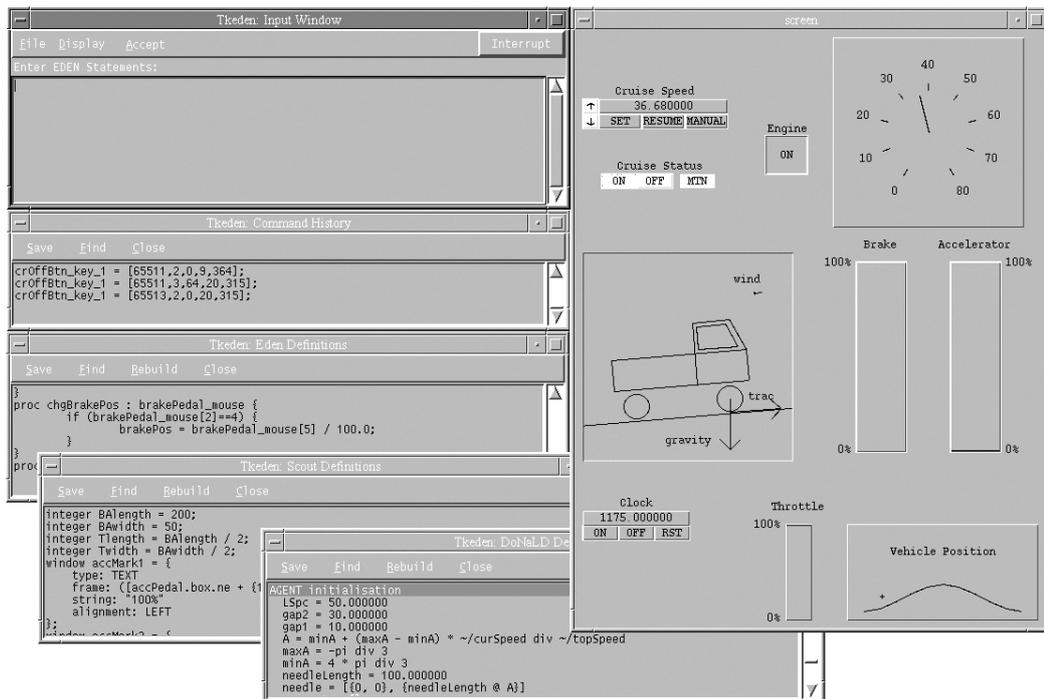


Figure 4.5: tkeden-dec151997 running *cruisecontrolBridge1991*

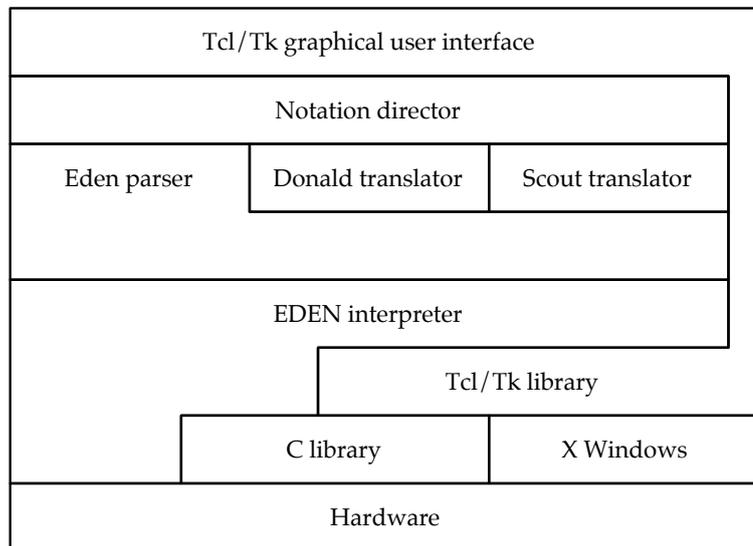


Figure 4.6: tkeden architecture

The approximate architecture of the original *tkeden*, viewed from a high level, is illustrated in Figure 4.6 (by this author). The graphical interface that can be seen in Figure 4.5, comprising the Input Window, the SCOUT screen and other windows, is implemented using Tcl/Tk. Input text is directed towards the appropriate translator which is selected by `%notation` directives in the input, as before. The DoNaLD and SCOUT translators are essentially the same as before: they translate input into Eden output, but the input/output mechanism has been changed. In the previous architecture, the translators were concurrent processes connected to the EDEN interpreter via a UNIX pipeline, but in this architecture, the appropriate translator routine is sequentially invoked with the input text and is expected to call routines to add Eden output to a queue for processing. Lower down in the figure, the EDEN interpreter calls the Tcl/Tk library routines in order to change the state of the screen. Tcl/Tk in turn calls the X Windows libraries to effect visual change.

4.1.6 Distributed *tkeden*: *dtkeden*

P-H. ‘Patrick’ Sun was the third person to undertake major work on EDEN, producing a variant named *dtkeden*, capable of distributed client-server communication of redefinitions. Sun describes the results in his PhD thesis [Sun99]. The distributed features in *dtkeden* are not examined in detail in this thesis as the analysis presented later on proceeds bottom up, starting from the sequential von Neumann machine hardware and the procedural C language in which the current EDEN tool is largely written. Chapter 5 of this thesis presents a framework for dependency maintenance which considers concurrency — future research will be required to reconcile this framework and *dtkeden*. It is probable that such a reconciliation will require a fundamental revision of the mechanisms for inter-process communication in *dtkeden*, which are currently implemented without formal attention to synchronisation.

The first *dtkeden* case study was *claytontunnelSun1999* — a model of the railway accident that occurred in the Clayton Tunnel on the Brighton to London Victoria line in 1861. The accident is described in Rolt’s “Red for Danger” [Rol82] and the case study is described in [Sun99] and [Bey99]. This model is again constructed using the Eden, SCOUT and DoNaLD notations. The *dtkeden* server shows “God’s eye view”, where all the observables are objectively presented — see Figure 4.7.

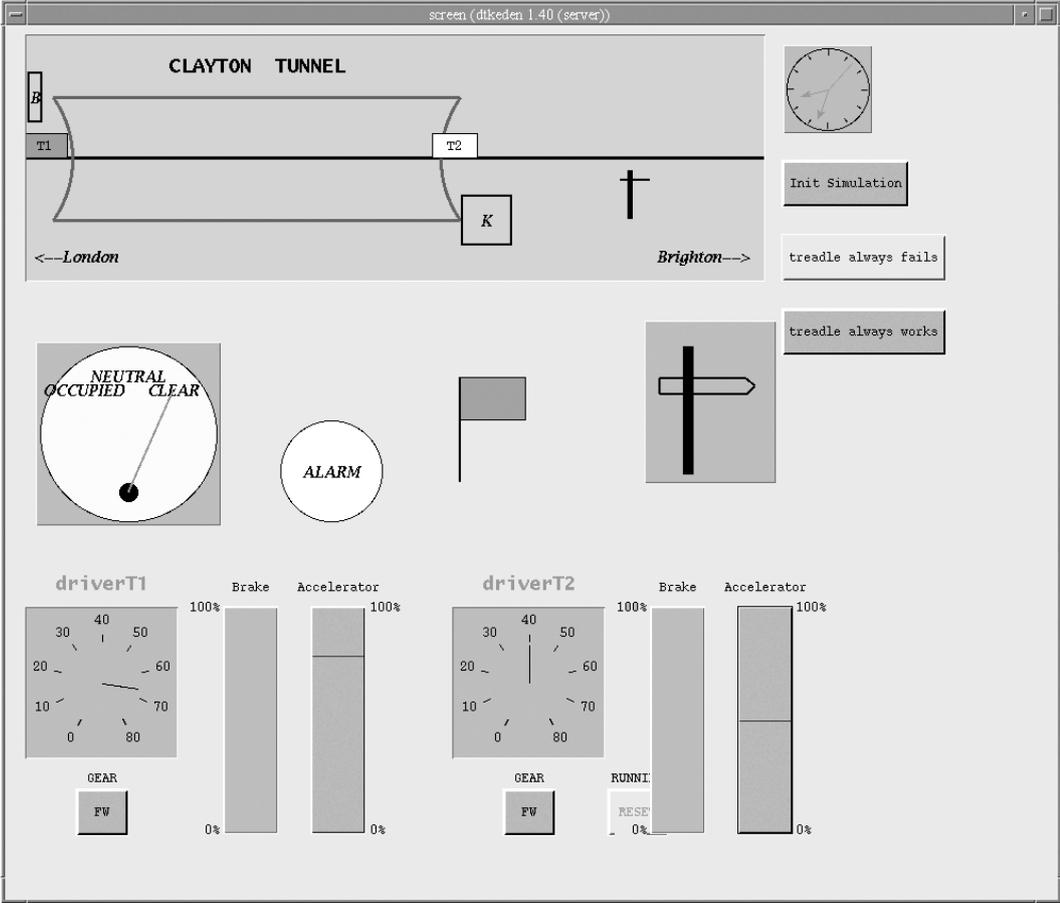


Figure 4.7: “God’s eye view” in *claytontunnelSun1999* on the dtkeden server



Figure 4.8: `dtkeden` and `claytontunnelSun1999` in use with school children during the ACE week in January 1999

The `claytontunnelSun1999` model requires five `dtkeden` client computers and human users, who play the roles of three train drivers and two signal men respectively, each seeing only their subjective perception of the situation. As part of our investigation into EM for educational applications, we introduced the model to several groups of school children in January 1999 (see Figure 4.8). As discussed by Roe [Roe03], the educational agenda has since motivated much model construction and the creation of the empublic archive [WRB].

4.1.7 Fifty versions: an overview

Figure 4.9 shows the growth of the EDEN code over the 17 year project period so far. The measure shown is a simple count of non-blank lines of code. An attempt has been made to avoid counting automatically generated sources (e.g. output from `yacc`). The count has been divided into that originating from C language files, Tcl files and Eden files⁹. All the source code that remains available has been measured, but only certain selected data points are shown.

⁹File types were determined as follows: files with `.c`, `.h`, `.y` and `.l` extensions (but not generated files) were counted as C language files; the `.tcl` extension counted as a Tcl file, and `.e` and the more recent `.eden` extension counted as Eden files.

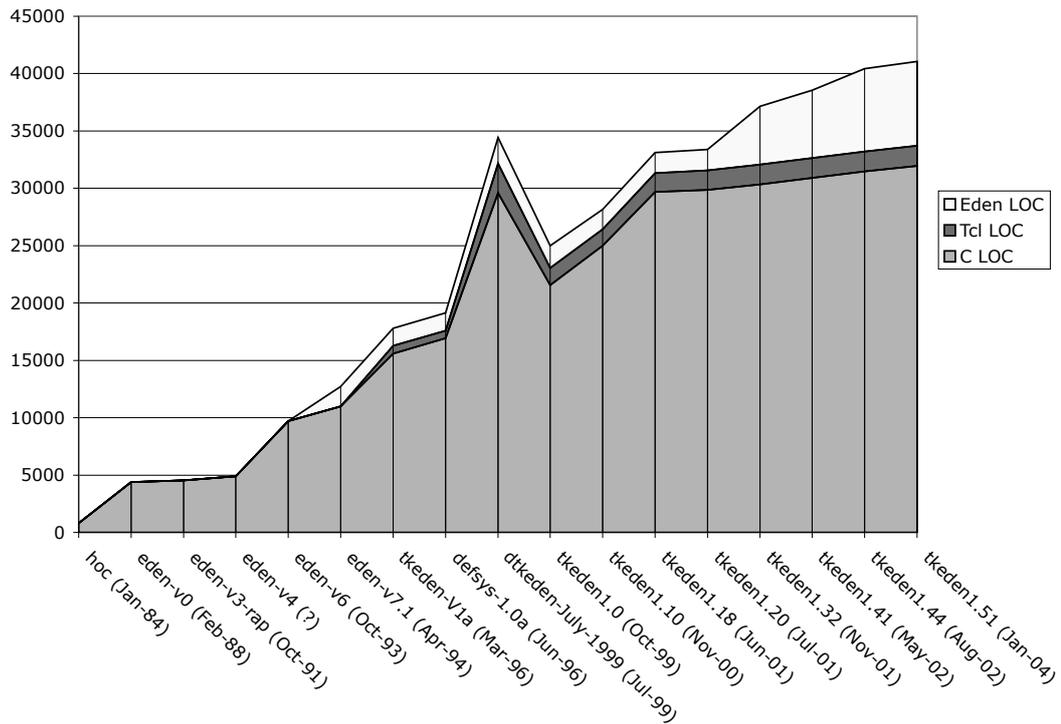


Figure 4.9: Code size of EDEN from 1988 to 2004 (1984 hoc shown for comparison)

The chart starts with the `hoc` source code for comparison and moves on to the earliest remaining EDEN source, `eden-v0` (which is actually not quite the earliest version as printed in [Yun87]). The EDEN sources from the version `v0` through `v7.1` to `tkeden-V1a` were developed by Y.P. Yung between 1988 and 1996. According to his documentation, it appears that the growth shown between `v4` and `v6` is due to merging of the EDEN/X interface into the main code base, creating `xeden`. The EDEN `todo()` command (see §4.3) was apparently added in `v6.1` and the SCOUT and DoNaLD translators were apparently integrated into `xeden` in `v7`: this can be seen in the chart as the introduction of 1725 lines of EDEN code in `v7.1`. The earliest remaining `tkeden` source code appears to date from March 1996 and includes 666 lines of Tcl code. The `dtkeden` data point is atypical when shown amongst the general progression. This version of the source has much duplication between variants of files for server and client. The duplication was reduced when I merged the `dtkeden` source code with the original `tkeden` source, making `ttyeden`, `tkeden` and `dtkeden` compile-time options from a single source. The remainder of the chart

illustrates changes and extensions since 1999, some of which are outlined below.

Reflecting on this period of computing history: the personal computing world was in its infancy in 1987. For example, the first Apple Macintosh was sold only three years before in 1984. Hennessy and Patterson [HP03, p.3] plot the growth in microprocessor performance from 1984-2000 and find that it has increased annually by 50%. The performance available now (in 2004) is therefore nearly 1000 times greater than was available in 1987.

An illustration of the effect of this change in performance on EDEN is given by Y.P. Yung and Farkas's model of a billiards game *billiardsYung1996*, which includes an element of simulation. The simulation uses an integration step-size that is adaptively dependent upon the ball speed [FBY93] in order to precisely determine the point of collision between balls. On the machines common in 1996, the simulation ran rather slower than real-time. The dynamic adaptation of the step-size also caused the ball velocity (as observed in the graphical model) to slow dramatically as another ball was approached. Y.P. Yung therefore added a facility to record the history of redefinitions as the simulation of a shot occurred, and later 'replay' it back, on demand. The replay of redefinitions is not computationally intensive and so can be linked to real-time, causing the replay to occur at a continuously realistic speed. Now in 2004, however, when run on a recent machine, we find that the initial billiards simulation occurs *faster* than real-time — the replay facility slows the simulation down to a continuously realistic speed. The continued progression of CPU performance, which is still currently following "Moore's law" [Moo65], leads to improvements in our models and tools on the basis of mere maintenance on our part. The continued progression is partial vindication of our prioritisation of meaningful state over raw performance. It seems likely that the performance increases will continue in the short term (to approximately 2009), but research breakthroughs are required in most technical areas if progress is not to stall in the long term [Wil02, AEWJ⁺02, itr].

This increase in performance has been accompanied by a growing desire to run EDEN on multiple platforms. Richard Cartwright first compiled the software for Linux. Ben Carter completed a port of `tkeden` to the Windows platform in early 2000. I have been maintaining these ports and also completed an initial port to

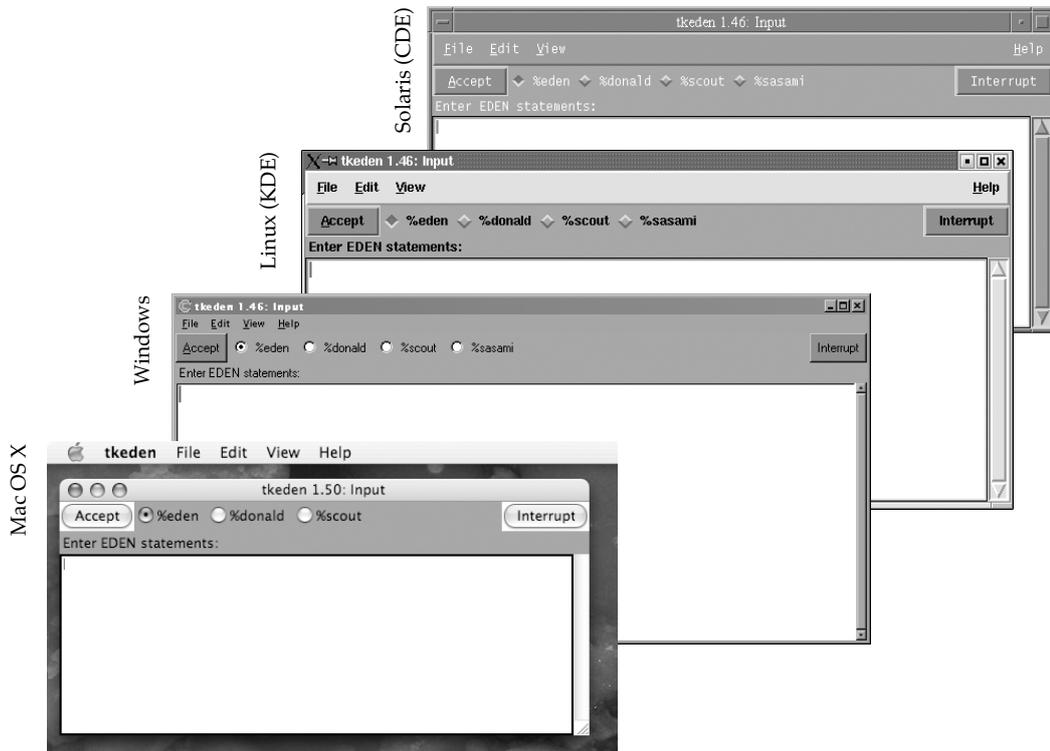


Figure 4.10: `tkeden` running on Solaris, Linux, Windows and Mac OS X

Mac OS X in early 2002, releasing a slightly more mature version in March 2003. Figure 4.10 shows the different appearances of the interface on the different platforms. In particular, the arrangement of the menus vary on each platform in order to conform to the differing interface guidelines on each.

Maintaining the tool on the different platforms makes developing the software more difficult, but it is a necessary part of attracting and retaining an audience for the tool.

The audience for the software has grown since the creation of EDEN. Presently, the vast majority of the EDEN user population is made up of students and staff within the University of Warwick Department of Computer Science. I set up and currently maintain an installation of the EDEN software for those local users. All previous versions are kept in the installation to aid regression testing. In November 2001, I added logging of software runs from the local installation. The time, date, user identity, host name, process identifier, tool variant and version, elapsed (wall

Month	Runs of EDEN	Unique users	Total CPU time (in hours)	Total CPU time / users (in mins)	Downloads via FTP				
					Win	Linux	Solaris	Source code	Docs
200111	141	42	17.2	24.6	23	1	0	0	2
200112	781	124	4.4	2.1	94	14	0	17	157
200201	124	9	64.6	430.5	3	0	0	0	0
200202	380	9	65.8	439.0	9	0	0	1	4
200203	229	9	20.5	136.4	9	0	0	0	5
200204	277	16	7.9	29.6	3	0	0	0	0
200205	355	11	4.9	26.6	21	3	2	6	9
200206	217	33	1.9	3.5	32	2	0	7	10
200207	85	4	0.5	7.1	7	0	0	3	5
200208	133	6	9.4	93.7	6	2	4	3	0
200209	69	5	0.6	7.4	5	1	0	0	7
200210	869	40	23.4	35.1	51	17	12	13	25
200211	3267	167	18.2	6.5	230	46	6	52	243
200212	248	19	5.9	18.6	12	8	3	8	49
200301	1826	16	3.5	13.1	19	0	0	1	4
200302	4044	19	25.0	79.1	3	2	0	2	0
200303	3588	20	61.0	182.9	4	2	0	1	2
200304	1892	13	9.4	43.6	531	98	27	114	522
200305	273	28	19.0	40.8					
200306	161	27	0.3	0.7					
200307	276	8	4.6	34.8					
200308	182	3	1.3	25.6					
200309	297	7	33.7	289.1					
200310	806	47	240.0	306.4					
200311	1280	109	174.6	96.1					
200312	503	18	5.9	19.7					
200401	1454	33	17.6	32.1					
	23757		841.2						

Figure 4.11: Data from logging of local tkeden usage and downloads

clock) time, user and system CPU time and the final exit status are all collected. As this logging is potentially intrusive, users are automatically asked to confirm their acceptance of this logging when they first run EDEN. The data collected is presented in Figure 4.11, summarised into months. The data shows that there is a continual background level usage of the tool, corresponding to use by staff and research students. The usage peaks when the software is used in undergraduate teaching: it has been used by many students as a basis for a third year project, on a large database systems level 2 module (producing the large peaks that can be seen in the numbers of unique users in November/December each year) and in an “Introduction to EM” level 4 module.

The usage data shown is usage on the local departmental systems only and does not reflect the additional usage of the software on users’ personal machines, which is increasingly popular. This is shown in Figure 4.11, which along with the usage

information, also includes information from logging of FTP downloads, summarised into counts of types of download per month¹⁰.

4.2 Developments to EDEN since 1999

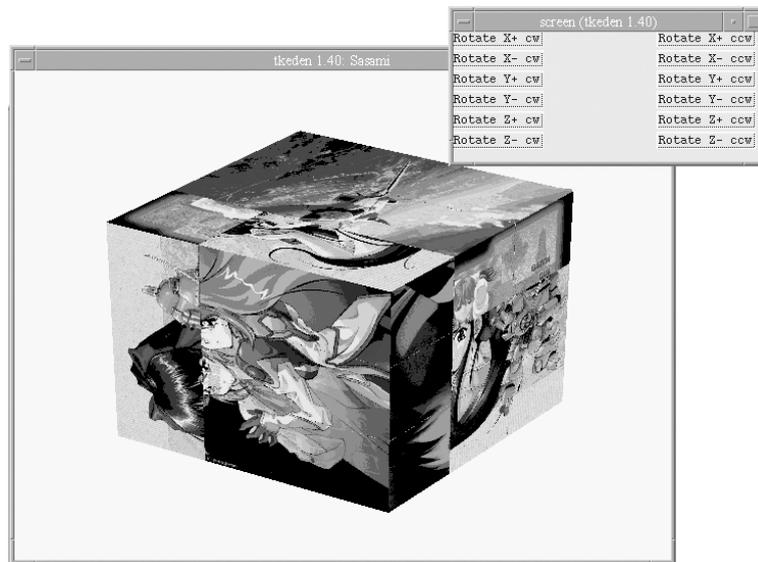
Since taking responsibility for the maintenance of `tkeden` in 1999, I have overseen some fifty released incremental versions of the software: approximately one release per month on average. This is in keeping with the open source “release early, release often” philosophy described by Raymond [Ray], which attempts to create an active feedback loop, assisting in testing the software and driving requirements. There have been many strands to the development, many taking significant effort, but it is not possible to give a full description here¹¹. The following material presents the most coherent strands. Subsections §4.2.1 to §4.2.5 inclusive relate to my aim of (re)constructing the tool “in itself”. Subsection §4.2.6 shows an environment I constructed in order to use the EDEN tool as a visual aid to support complete presentations, including multiple demonstrations. This environment incidentally made it obvious that multiple models can be ‘run’ simultaneously in one instance of EDEN. Subsection §4.2.7 briefly describes improvements I have made to the EDEN interface. The subsection includes a description of a definitive partial reconstruction of the `tkeden` interface by Roe, within the presentation environment. The reconstruction is another interesting “in itself” prototype. The final subsection §4.2.8 describes several experiments we have performed that interface EDEN to other systems. The experiments show that dependency can be used successfully in this manner, but issues of concurrent synchronisation are raised — in part due to the single-threaded sequential nature of the EDEN machine, which is explored in the final section §4.3.

4.2.1 The beginnings: Sasami, motivating “in itself”

Figure 4.9 shows that in 2001, a decision was made to prefer to extend the tool by adding Eden code over C code. As of version 1.51, the Eden code constitutes 18%

¹⁰The ability to use HTTP downloads was added around March 2003, providing a choice of download facility but making the FTP statistics invalid for comparison, and so they are not shown from this point onwards.

¹¹For such details, consult the ‘change log’ file at [Wara].

Figure 4.12: The Sasami Rubik’s cube, *rubiksCarter1999*

of the total. The motivation for this decision originates with Sasami.

Sasami was the fourth notation (additional to Eden, DoNaLD and SCOUT) to be integrated in the `tkeden` tool. It is a notation for 3D graphics and is implemented using the currently standard OpenGL 3D graphics library. Sasami is the final year project work of Ben Carter [Car00], who was also responsible for the initial `tkeden` port to Windows. The first sizeable case study was *rubiksCarter1999*: a 3D model of Rubik’s cube. Figure 4.12 shows the cube after some rotations have been applied using the buttons in the SCOUT interface window at the top right.

The Sasami implementation is technically similar to the DoNaLD and SCOUT implementations. A translator converts statements written in the Sasami notation to Eden code. The Sasami notation is a simple one which is similar to the DAMscript code described in §3.3.2. Parsing it requires no semantic analysis, and so the translator is implemented directly in C rather than with the yacc parser generator. For example, the basic primitive in Sasami (as in OpenGL) is the coplanar convex polygon, which is described by a list of vertices (points in Cartesian 3D space). To create a Sasami vertex which can later be used in a polygon vertices list, the `vertex` definition is used. Such a definition has no explicit `is` primitive; it takes the form of the keyword `vertex` and a symbolic name, followed by `x`, `y` and `z`

```

v=1;
_sasami_vertex_1_x is (a+10)/20;
_sasami_vertex_1_y is -c;
_sasami_vertex_1_z is 1.0;
proc _sasami_vertex_mon_1 :
    _sasami_vertex_1_x,
    _sasami_vertex_1_y,
    _sasami_vertex_1_z {
    sasami_vertex(1,_sasami_vertex_1_x,
                  _sasami_vertex_1_y,
                  _sasami_vertex_1_z);
};

```

Listing 4.8: EDEN output corresponding to the Sasami definition `vertex v (a+10)/20 -c 1.0`

coordinates. Each coordinate is described by an Eden expression. For example, the following is a valid Sasami definition:

```
vertex v (a+10)/20 -c 1.0
```

Sasami notation is translated to Eden code. At low levels, Sasami uses unique integer identifiers (which are incremented each time a Sasami object is created) rather than symbolic names. The Eden code therefore contains some indirection from symbolic name to a numeric identifier. Otherwise, the translation is similar to that done by the DoNaLD translator: definitions are created to represent dependencies between objects described at the Sasami level. An Eden action is then created for each Sasami object in order to mediate changes of definitive state to the graphical display. The Eden action calls built-in functions added to EDEN especially for Sasami. For example, the Sasami `vertex` command above is translated into the Eden code shown in Listing 4.8.

The Eden action here calls the new `sasami_vertex` EDEN built-in. This is another part of Sasami, corresponding to the Tcl/Tk graphics subsystem used by DoNaLD. This part of Sasami contains a store of objects referenced by numeric identifiers. Sasami routines call OpenGL routines when rendering is required and read from this store. The data flow is illustrated in Figure 4.13¹².

¹²When porting Sasami to UNIX in version 1.13, I introduced a delay in the bottom-most edge shown in the figure. Now, the graphics hardware is only invoked at ‘Tcl.Update’ (see Figure 4.38), effectively grouping multiple changes of Sasami state into blocks.

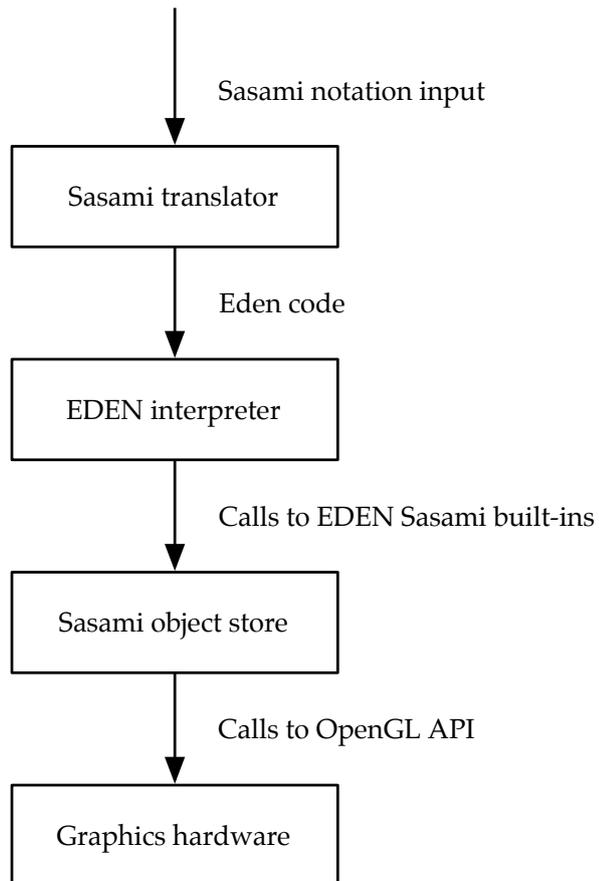


Figure 4.13: Data flow in Sasami



Figure 4.14: The effect of changing the `small_cube_size` variable in *rubiks-Carter1999*

The rotations performed on the Sasami Rubik’s cube operate by reassigning cube coordinates and so do not demonstrate the use of dependency¹³. However, the model does contain the Eden variable `small_cube_size`, which controls the size of the distinct cubes through the use of dependency. Figure 4.14 shows the effect of redefining the value from 0.5 to 0.1.

OpenGL is a cross-platform standard but it does not provide an abstraction of the window system. Initially, this part of Sasami was coupled to the Windows platform. In version 1.13, I reengineered this part using the open source ‘Togl’ package, which provides an OpenGL Tcl/Tk widget. This allowed Sasami to operate on UNIX as well as Windows¹⁴ and should allow integration of the Sasami window into the SCOUT environment in future.

Sasami has now been used in several models (*sasamiexamplesCarter1999*, *rubiksCarter1999*, *billiardsCarter1999*, *room3dsasamiCarter1999*, *cubesym-Wong2001*, *platonicSolidsBirch2001*, *3doxoRoe2001*, *pyramixRoe2001*, *carparking-simMcHale2003*). Although these models have been successfully constructed, each has encountered problems with instantiating multiple instances of similar state. This has been solved in each case by writing a program that generates model code. Sometimes the generator program is written in C. In other instances, the generator program is written in Eden and calls the `execute()` command to instantiate state. Either solution presents problems when it is desired to make redefinitions to the model later. This problem seems especially acute when using Sasami. The notation may well be too low-level: it does not have the algebraic properties of DoNaLD or the (unimplemented) definitive notation CADNO (Computer-Aided Design Notation — for which a preliminary design is proposed in [Bey89b]).

Adding Sasami in version 1.7 increased the number of lines of C code, as Figure 4.9 (on p.219) shows. Adding functionality to the EDEN tool by extending the procedural code in this way causes several problems. The following problems are symptomatic of compiled procedural languages generally:

- The tool needs to be recompiled when changes are made. Although this process is faster than in 1987, it is still slow.

¹³It may be possible to model rotations of the individual cubes using dependency by modelling the internal armature mechanism (the ‘core’ — see [Cao02]) of the cube.

¹⁴Work on porting Togl to Mac OS X is underway.

- Once the code is compiled, it is then fixed. The audience cannot then modify the tool if it does not suit their needs.
- It is difficult to fully test the result. Ideally all possible behaviours would be exercised (black-box testing). This is clearly an unbounded problem in a general modelling tool. A smaller aim would be to exercise all the lines of code (white-box testing). This requires reverse engineering the control flow into a set of test cases which provides complete code coverage. The tests must be revised when the control flow is modified.
- Modifying the procedural code has the possible side effect of changing the behaviour of the tool. This might then cause some previous models to behave differently, and we have broken backwards compatibility.

Additionally, although C has many advantages when implementing a low-level virtual machine (see §4.3.5), it has disadvantages when used for constructing a translator string processor:

- String handling in C requires explicit management of memory allocation and deallocation.
- C does not (by default) perform range checks on buffer accesses.

Both of these attributes of C have led to problems in EDEN that were difficult to locate and resolve. Checks for problems with memory management can be made automatically by linking with a debugging memory allocator library (for example, ‘dmalloc’ [Wat]), and I have used this technique to assist in debugging EDEN. However these techniques cannot be used routinely as the debugging allocator greatly slows down execution.

Further, modular construction leads to multiple copies of identical state within different modules. The three copies of Sasami state shown in Figure 4.13 — one copy in the EDEN symbol table, one in the Sasami object store and one in the OpenGL machine are one example. In the Sasami case, this is a waste of memory, which may or may not present a problem. The current DoNaLD system presents a similar example with what might be considered worse consequences. The DoNaLD notation

is strongly typed. However, the strongly typed facilities of DoNaLD are somewhat contrary to our needs today when we consider the “incremental construction and revision” motivation for our tools. As a consequence of strong typing, the DoNaLD translator maintains its own symbol table (as shown in Figure 4.3), in order to perform type checks on DoNaLD expressions and resolve overloaded operators. But the DoNaLD symbol table is inaccessible from Eden, limiting flexibility and leading to inconsistency if one changes DoNaLD state that is stored in EDEN.

EDEN however determines types of variables automatically. The massively increased CPU performance now available makes it more viable today to perform more of the definitive notation translation task in Eden, leading to flexibility gains and potentially less memory usage.

There is another motivation for performing definitive notation translation in Eden. The integration of notations into the single `tkeden` tool removed the shortcomings noted by Y.P. Yung (§4.1.5): it is possible for EDEN to generate DoNaLD or SCOUT definitions and cause them to be parsed using the `execute()` command, and the integrated system has led to an improved user interface, making the tool more accessible and allowing the research group to focus on applications. However it gave the integrated notations, SCOUT and DoNaLD, undue prominence, making it difficult to use other also interesting notations such as ARCA [Bey83], CADNO [Bey89b] and Sand (*sandSocket1992*). The integration also limited the possibilities for experimentation with creating and modifying definitive notations.

The preceding paragraphs, then, have given the motivation for implementing EDEN “in itself”. The following sections describe progress related to this theme.

4.2.2 A database in EDEN: EDDI

In 1996, S.V. Truong created the EDDI (EDEN Database Definition Interpreter) notation [Tru96]. It is a definitive relational database language based on ISBL [Tod76]. The syntax is shown in Figure 4.15.

The syntax used to define views in EDDI (which, in Figure 4.15 includes the “define view” command and the syntax listed below it) can be considered to make up a pure definitive notation. It can thus be translated into Eden definitions. The procedural statements (above the “define view” command) can be translated into

COMMANDS	create table	<i>table_name (attribute_name, attribute_type, attribute_name, attribute_type...);</i>
	assignment	<i>table_name = relational_expression;</i>
	insert tuples	<i>table_name << [attribute, attribute], [attribute, attribute], ...;</i>
	delete tuples	<i>table_name !! [attribute, attribute], [attribute, attribute], ...;</i>
	truncate table	<i>~table_name;</i>
	drop relation	<i>~~relation_name;</i>
	describe	<i>??relation_name;</i>
	query	<i>?relational_expression;</i>
	show catalogue	<i>#; (or ?CATALOGUE;)</i>
	define view	<i>view_name is relational_expression;</i>
OPERATORS	union	<i>X + Y</i>
	difference	<i>X - Y</i>
	intersection	<i>X . Y</i>
	join	<i>X * Y</i>
	selection	<i>relational_expression : predicate</i>
	projection	<i>relational_expression % attribute_name, attribute_name...</i>
	project and rename	<i>relational_expression % attribute_name >> attribute_name, attribute_name >> attribute_name...</i>
PREDICATES	equality	<i>attribute_name == attribute_name</i>
	less than or equal to	<i>attribute_name <= attribute_name</i>
	greater than or equal to	<i>attribute_name >= attribute_name</i>
	not equal to	<i>attribute_name != attribute_name</i>

Figure 4.15: EDDI syntax

procedural Eden statements.

Truong created two implementations of EDDI based on `ttyeden`. EDDI/R (*ed-dirTruong1996*) was the ‘real’ implementation, where `ttyeden` was connected to an Oracle database by extending EDEN with functions written in Pro*C containing embedded SQL. EDDI/R has not been seriously used by anyone except Truong and is not discussed further here.

EDDI/P (*eddipTruong1996*) was a ‘pseudo’ implementation of EDDI. In EDDI/P, the relational data is actually stored as lists in the EDEN symbol table, rather than in an Oracle database, hence the ‘pseudo’ nomenclature. EDDI/P is a translator implemented as UNIX filter, designed to pipe into `ttyeden`, for example using the command:

```
cat eddipf.e - | eddip | ttyeden -n
```

The example given in Listing 4.9 shows EDDI/P in use, and also the (normally hidden) translator output. Textual formatting is here used to distinguish between `the EDDI input`, *translator Eden output* and the `output to the user`. Note that utility functions written in Eden (e.g. `create`, `project`, `inter`) are used in the translator output in much the same way that the DoNaLD translator uses EDEN procedures to create geometric objects and perform operations on them (*cf.* §4.1.4). In EDDI, these utility functions are defined in the `eddipf.e` Eden support library file loaded as a part of the UNIX pipeline above.

The EDDI/P translator together with `ttyeden` was provided by the author and Beynon as a student resource to assist with the teaching and learning of relational algebra in a database systems module at Warwick, originally in 1999 and again in 2000 (*eddipWard2000*).

4.2.3 A front-end parser in EDEN: the Agent Oriented Parser

The original EDDI system, being based on `ttyeden`, provided only terminal-based interaction with relational algebra. Using `tkeden` would provide an improved interface and it was thought that potentially the graphical facilities of `tkeden` could be used to provide visualisation of the database. However, it was not possible at that time to pipe input into `tkeden` and, because of the reasons stated above, it was desirable to avoid adding the EDDI/P translator code, written in C, to EDEN.

```

%eddi
FRUITS (NAME char key, BEGIN int, END int);
  FRUITS = create("NAME", "#", "BEGIN", "", "END", "");
FRUITS << ["granny",8,10],["lemon",5,12],["kiwi",6,7],;
  FRUITS = addvals(FRUITS,["granny",8,10],
    ["lemon",5,12],["kiwi",6,7]);
?FRUITS;
  showrel(FRUITS);
-----
NAME          BEGIN          END
-----
granny        8              10
lemon         5              12
kiwi          6              7
-----
NAMES is FRUITS % NAME;
  proc _rt2: FRUITS { _re2 = project(FRUITS,["NAME"]); };
  NAMES is _re2;
?NAMES;
  showrel(NAMES);
-----
NAME
-----
granny
lemon
kiwi
-----
SUMMERFRUITS is ((FRUITS: BEGIN >= 6) . (FRUITS: END < 9)) % NAME;
  proc _rt5: FRUITS { _re5 = select(FRUITS,"BEGIN", ">=",6); };
  proc _rt7: FRUITS { _re7 = select(FRUITS,"END", "<",9); };
  proc _rt8: _re5,_re7 { _re8 = inter(_re5,_re7); };
  proc _rt9: _re8 { _re9 = project(_re8,["NAME"]); };
  SUMMERFRUITS is _re9;
?SUMMERFRUITS;
  showrel(SUMMERFRUITS);
-----
NAME
-----
kiwi
-----

```

Listing 4.9: Interaction with the eddip translator

A method of replacing the `eddip` pipeline translator was devised; it makes use of what is now called the Agent-Oriented Parser (AOP). The AOP is written mostly in Eden (with a few small changes required in the EDEN C code to arrange for input to be passed to the Eden interpreter) and is capable of parsing many languages. More specifically, however, first we consider the AOP as a black box translator, parsing EDDI input. The upper part of Figure 4.16 shows the AOP translation of a line of EDDI input to several Eden definitions.

The AOP EDDI translator creates several lines of EDEN output for one line of EDDI input. The several Eden definitions that are created are each of a relatively simple nature. In this respect, the AOP EDDI parser resembles the DoNaLD to DAMscript compiler described in §3.3.2. Sub-expressions of the original input can be observed: for example, `var_60` in Figure 4.16 holds the value of the sub-expression `CITRUS % NAME`. This value could potentially be re-used if another definition required the same sub-expression, reducing the total amount of re-evaluation time. Parts of the original expression can also be modified without a need to invoke the parser again.

Although the mapping of a single EDDI definition to many Eden definitions described has these advantages, the result can be inconvenient when EDDI and Eden are being used in close conjunction in a complex EM exercise. In her final year project [Oun04], Asma Ounnas has written Eden procedures to recursively modify the script graph that results from an AOP EDDI translation, transforming the structure into a form that is isomorphic with that of the input. The modification is possible due to the simple “referentially transparent”¹⁵ nature of a set of definitions. Structural changes such as these can modify the amount of internal detail in a script graph, whilst leaving the maintained relationships between leaves and roots unchanged. Although the post-processing technique shown in Figure 4.16 seems inefficient, it provides more development options for the modeller — the removal of internal detail may or may not be significant.

Thus far, the AOP has been discussed as a black box. The following is a brief overview of the internal operation of the AOP and its subsequent development and use. It illustrates a pattern of development through progressive construction by a

¹⁵I place the term in quotes as referential transparency in a definitive script applies only within an unchanging state.

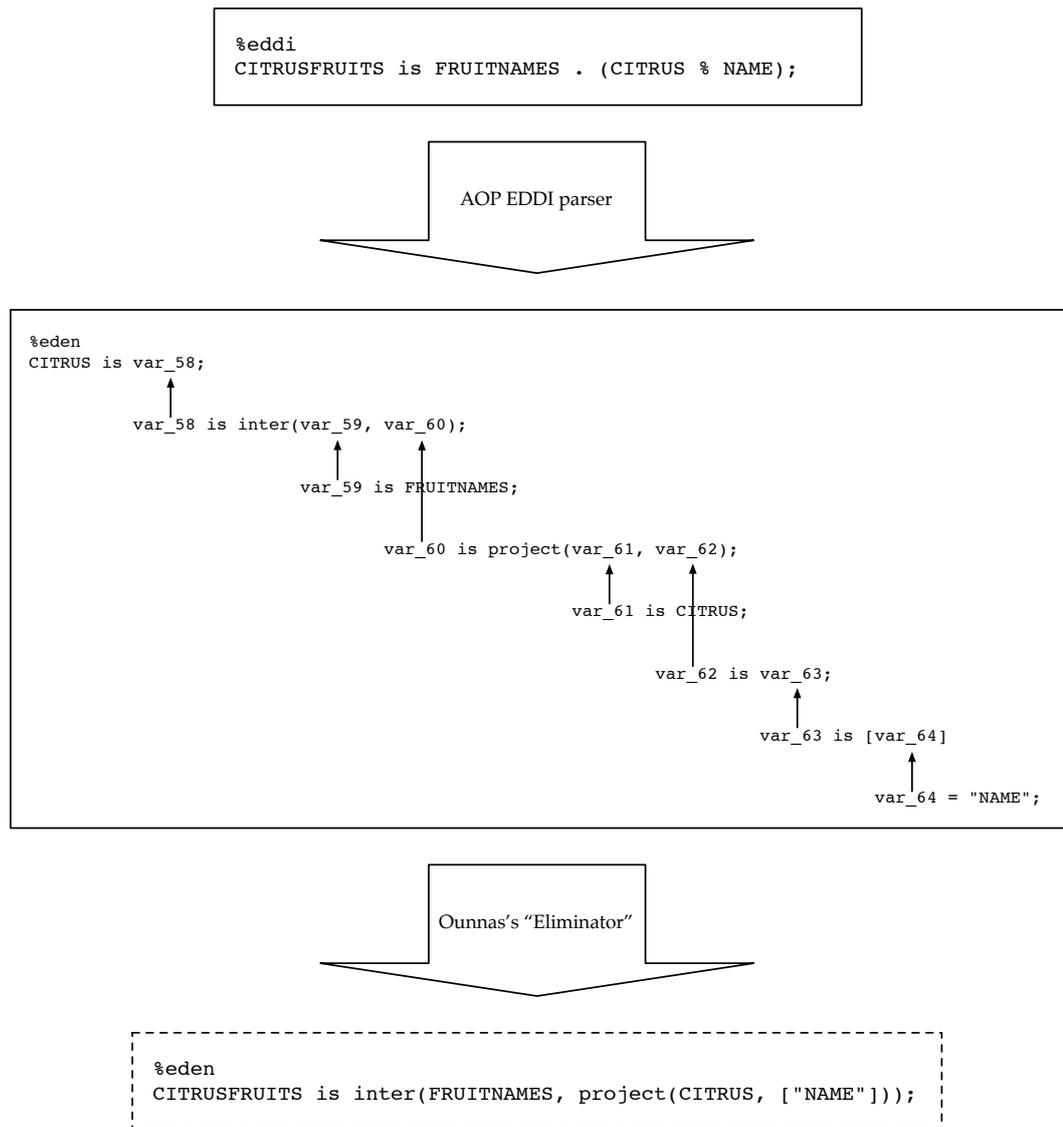


Figure 4.16: AOP EDDI to EDEN translation

number of varied authors which is not unusual in EDEN model development, to which my work has made a crucial contribution.

The AOP is originally the work of Brown (*agentparserBrown2001*). He wrote AOP configurations that enabled the parsing of several types of input, including palindromes, the procedural language PL/0¹⁶ and EDDI. Brown's EDDI parser used the `eddipf.e` Eden support library file originally developed by Truong (*ed-dipTruong1996*). Evans and Brown created an SQL AOP parser configuration in September 2001, which I included in version 1.26 of the EDEN distribution. Beynon improved the SQL parser (*sqleddiBeynon2001*) and used it as the successor to EDDI/P when teaching the Warwick database systems module in November 2001. In February and March 2002, Roe produced a parser configuration and interactive environment for the LOGO language (*logoparserRoe2002*) and constructed a simple clown-control notation constructed on top of LOGO (*krustyRoe2002*). In August 2002, I added regular expression processing facilities to EDEN. Harfield simplified the AOP in May 2003 (*agentparserHarfield2003*), incorporating regular expressions to parse literals. He also documented the AOP and wrote parser configurations for a simple calculator notation similar to `hoc` (§4.1.1) and a parser configuration for a systolic array definitive notation (*sandHarfield2003*), originally developed in C++ with `lex` and `yacc` support by Sockett in 1992 (*sandSockett1992*). In July 2003, Roe and I developed a presentation for a workshop on Teaching, Learning and Assessment in Databases [BBRW03], using EDEN to generate PowerPoint-like slides, some containing live demonstrations of EDDI (see §4.2.6).

As an indication of the nature of the work that was involved in supporting the above development, in version 1.43, I generalised the notation handling framework (see §4.2.4) and added regular expression processing facilities to EDEN. The use of regular expressions is well documented in Fried's "Mastering Regular Expressions" [Fri97]. For implementation, rather than use an operating system library for regular expressions and risk incompatibilities between EDEN tools running on different platforms, I chose to use Hazel's "Perl-compatible regular expressions" package [Haz]. I used the resulting facility to construct an Eden preprocessor `'%edens1'` in an attempt to ease problems of using Eden lists definitively (see §5.2.1).

¹⁶Introduced by Wirth [Wir76].

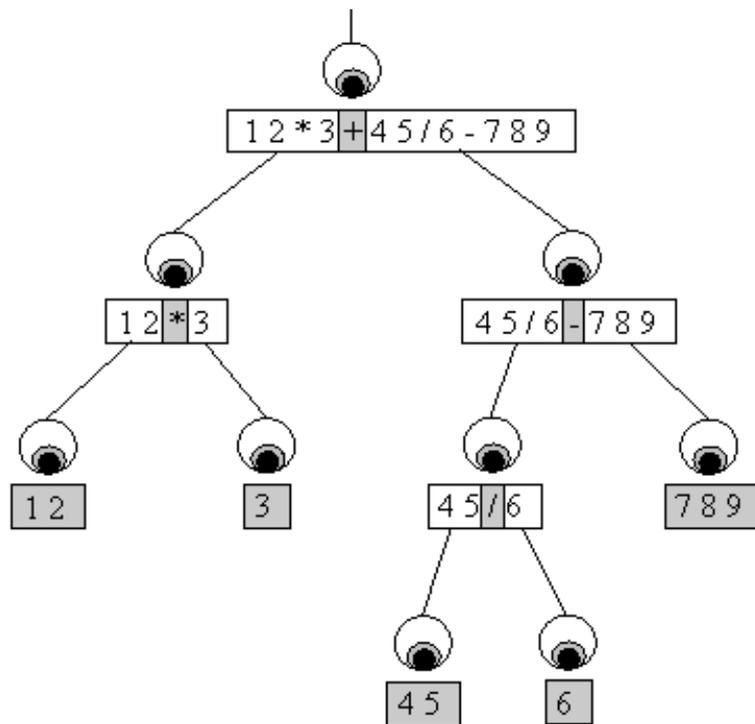


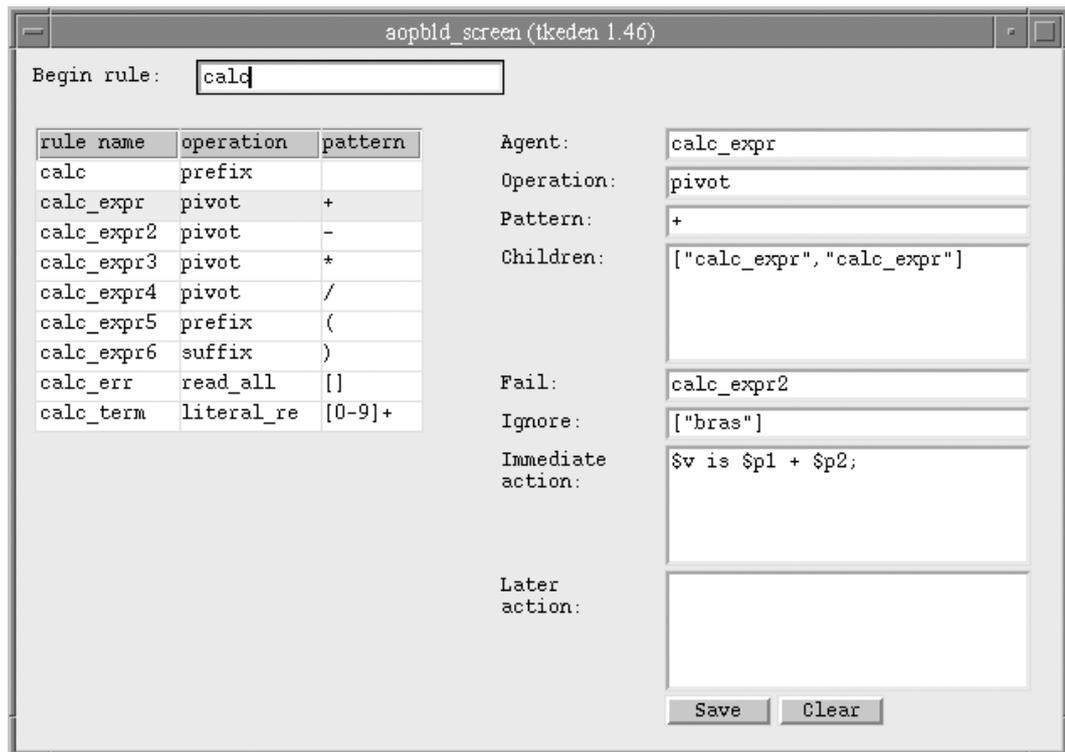
Figure 4.17: Observational structure of the AOP (from *agentparserHarfield2003/docs/tutorial/*)

The AOP is used to construct a parser whose operation can be understood using the EM concepts of observation, dependency and agency. Initially, an ‘agent’¹⁷ observes the entire input string in an attempt to locate the token considered most significant in the definitive notation being parsed. If the token is found, the agent creates a definition whose sources are initially undefined (@), and then instantiates child agents to generate definitions for the remaining sub-sequences of the input (see Figure 4.17). The method of repeatedly sub-dividing the input string leads to three interesting consequences:

- The termination of the parser is in principle guaranteed, since the primary activity involved is sub-division of the input string¹⁸;

¹⁷This term is used as a reference to the ADM and LSD sense of the word but this sense is lost somewhat in the EDEN implementation.

¹⁸Operator precedence is currently handled by the automatic insertion of parentheses around the highest precedence operators and operands, and this can only extend the length of the input by a bounded amount.

Figure 4.18: Harfield's parser builder in *agentparserHarfield2003*

- When a parse error occurs, the erroneous substring can be provided to the user for correction, rather than the entire input;
- Similar to definition maintenance (see §5.1), the child agents could in principle execute in parallel, or a depth- or breadth-first implementation can be chosen. The original author, Brown, experimented with depth- and breadth-first implementations — the current EDEN does not support concurrent execution.

The instantiation relationships between agents are described in an AOP parser configuration, where the notation grammar and parser agent actions are encoded into Eden lists. Harfield has built EDEN models using DoNaLD and SCOUT to show graphically the parser configuration and also how a particular input is parsed: see Figures 4.18 and 4.19 respectively.

The parser configuration can be changed interactively whilst the EDEN system is running. However this does not currently lead to a re-parsing of the input.

As the parser configuration is represented as Eden lists, it can be made dependent

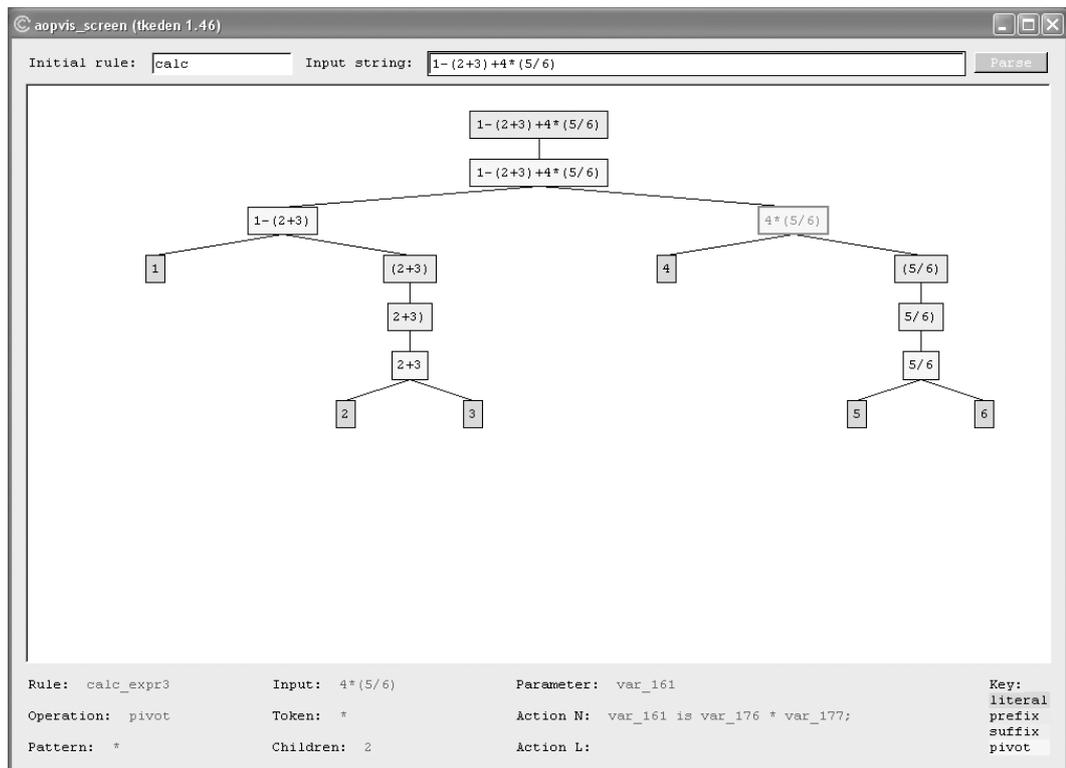


Figure 4.19: Harfield's parser visualisation in *agentparserHarfield2003*

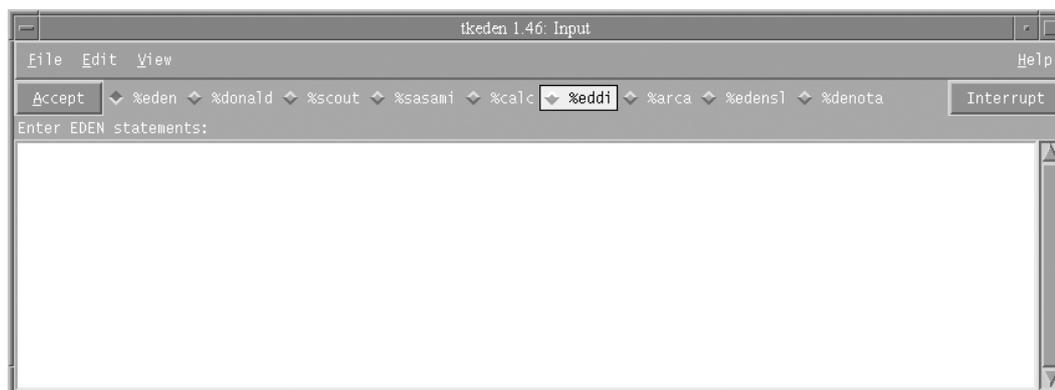


Figure 4.20: `tkeden` input window with a selection of the multiple notations now available installed

on other variables. Brown uses this idea in his palindrome parser, where a parser configuration causes the prefix and suffix of the input to be repeatedly read and checked for equality. Dependency is used within the parser configuration in such a way that the suffix will always be checked with the last read prefix.

The available parser configurations are made visible in `tkeden` through facilities I added in version 1.43 to create the necessary radio buttons when a notation is installed: see Figure 4.20, which shows the tool with many notations installed and ready for use.

4.2.4 Generalising the parser solution: the notations framework

A further improvement implemented in version 1.43 of EDEN is the generalised notations framework. Notations need not be implemented by writing an AOP parser configuration: if this is more convenient, they can be implemented directly in Eden, which can call into functions written in C if required. Installing a new notation is done by in EDEN by calling an Eden procedure `newNotation`, which requires the following parameters:

```
newNotation(name, switchProcPtr, transProcPtr);
```

The first parameter provided here is the name of the notation implemented. The second parameter is a pointer to a `switchProc` procedure, which will be called (in a form of ‘call-back’) whenever the Notation Director (see Figure 4.6, p.215) detects that the input mode has switched to the notation `%name`, allowing initialisation of

the parser. The last parameter is a pointer to a `transProc` procedure which will be called once for every character read from the input. This procedure is expected to translate the input and output the according Eden code using calls to the EDEN `execute()` command. The name of the notation is provided when the `switchProc` and `transProc` procedures are called and so the procedures can be shared amongst multiple notations (as they are in AOP notations, for example).

Notations installed in this way take precedence over the built-in notations and so can replace the existing DoNaLD, SCOUT and Sasami translators (which are written in C with `lex/yacc` support) or even replace the default Eden notation. This feature should allow improvement of the basic notations in the tool without the need for such a project to involve recompilation of the tool or knowledge of the C source code.

Augmentation of the existing notations is also possible as aliases for the existing notation names can be introduced. For example, the existing built-in notation `%donald` has an alias¹⁹ `%donald0`. A new notation named `%donald` can therefore be introduced using `newNotation()`, and the implementation can generate `%donald0` output for cases that it does not wish to handle.

Notations generating output in notations other than Eden leads to the need for each parser to be re-entrant. This is an issue if, for example, a notation `%one` that is implemented using `switchProc1` and `transProc1` generates output which calls a notation `%two` that is implemented using the same procedures. The KRUSTY notation (*krustyRoe2002*) is dependent upon the LOGO notation (*logoparserRoe2002*) in this way as they are both implemented using the AOP. Work on the system to implement a re-entrant parser is near completion.

4.2.5 Making the parser dependency driven?

Given that the AOP is written in Eden, it may be possible to control the parser using dependency. That is, to make the Eden output from the parser dependent upon the input text to the parser. As described in §4.1.4, definitive notations are

¹⁹In version 1.43 and later — DoNaLD only at present.

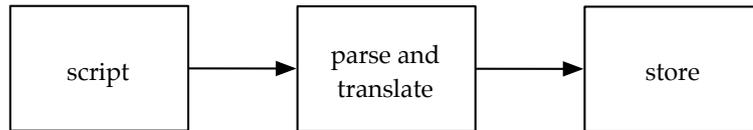


Figure 4.21: Notation input data flow in EDEN

generally implemented in EDEN by construction of a translator parser that converts definitive notation script input into Eden language output, which is then executed or stored by EDEN as appropriate. The data flow is shown in Figure 4.21. Presently, the data flow illustrated in Figure 4.21 is controlled in a conventional way: the entry of script input causes a procedure call to the translator with the portion of script passed as a parameter. The translator then uses further procedure calls to output Eden code and change the underlying state of the EDEN machine. The procedure calls form a ‘one-off’ process and little information about the process is retained after it has completed. Pursuing the “in itself” motivation, it would seem possible to construct dependencies for the edges in Figure 4.21, controlling the translator using dependency.

A dependency-driven parser would require that the EDEN system store all currently valid input (the ‘script’ input) in its original form²⁰, to provide input to the parser dependency. The dependency structure constructed would allow Eden code in the system to be traced back to the corresponding original input. (In the current EDEN, this is not possible without knowledge of translator conventions.) Changing the input text would cause it to be automatically re-parsed. Further, changing the parser itself would also cause the input to be automatically re-parsed, maintaining and guaranteeing the definitive relationship described by Figure 4.21.

Figure 4.21 actually describes a many-to-many mapping, since the script and store are both non-atomic areas of memory. Given a foundation capable of maintaining such a definitive mapping, it would be possible to make *sections* of the store dependent on *sections* of the script, in such a way that (for example) changing one character of the input would lead to the minimum necessary re-parsing.

²⁰The input is in fact stored by the current EDEN system, but in a somewhat superfluous way: the only purpose it serves is for user observation in the interface (see the Eden Definitions window in Figure 4.5 on p.215).

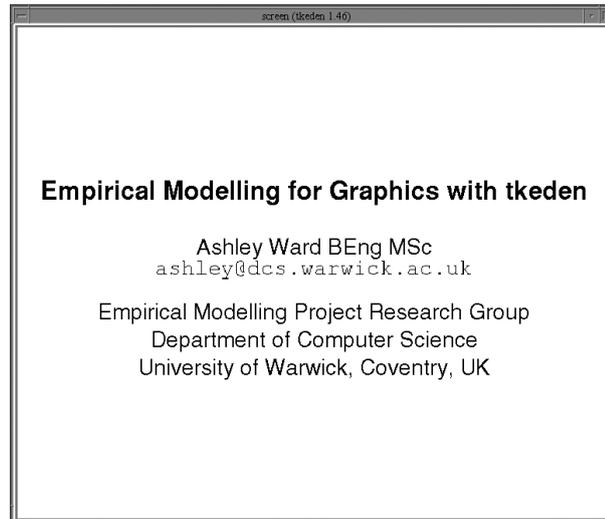


Figure 4.22: The first presentation environment slide

It is also interesting to note that the processing of the Eden language itself can also be described by Figure 4.21 — the EDEN implementation is a translator parser that generates EDEN VM code (see §4.3.5). Controlling this translation with dependency would be a further challenge.

4.2.6 Demonstrating EDEN with presentations

Prior to 2001, demonstrations of the EDEN tool usually took the form of an introductory talk supported by OHP or PowerPoint slides, followed by a short ‘case study’, showing EDEN and one or two models. Inspired by a demonstration of “Imagine” [KB00] (an OO LOGO tool), I created a basic presentation environment for EDEN, using it first for my presentation at the Libre Software Movement (LSM) conference, Bordeaux in 2001 (*lsmpresentationWard2001*). My initial slide is shown in Figure 4.22.

I attempted to make it possible to use dependency as much as possible in the presentation environment. To enable this, I extended and improved the underlying EDEN tool in various ways. For example, the text shown in the slide happens to all be centred horizontally across the window. If the window is resized, the text repositions in order to remain centred in the window. The text repositioning was achieved by extending EDEN to include a `screen_width` variable and by rewriting

```

titlef = "{helvetica 27 bold}";

s101 = "Empirical Modelling for Graphics with tkeden";
s101f is titlef;
s101w is StringWidth("screen", s101f, s101);
s101x is (screen_width - s101w) / 2;
s101y is 200;

```

Listing 4.10: EDEN script corresponding to part of Figure 4.22

an existing `StringWidth` function (initially written in C) in Eden. These two features allow dependencies in the form shown in Listing 4.10 to be created. Changes to the text, font or window width then propagate and cause repositioning of the on screen text.

The x coordinates of the text are therefore calculated by dependency in order to centre the text on the screen. The y coordinate of the first line of text on the screen is given using a literal value, but subsequent lines of text are positioned with reference to the height and position of lines of text above, again using dependency in a similar way.

This first slide has six items of text, some with different fonts and line spacing. In an attempt to make slide editing easier, I used Y.W. Yung’s macro function [Yun90, Appendix B] to perform pre-processor-like substitution on a script template, which is then instantiated using the Eden `execute()` function. This is analogous to an instantiation of an object from an OO class template, using a constructor. I wrapped this in an Eden “string, centered” procedure named `sc()`. The corresponding EDEN script that configures the slide is shown in Listing 4.11.

As well as text, a slide in the presentation environment can also contain an existing Eden model. I included the ‘jugs’ model (*jugsBeynon1988*, [BY90]) in a slide, and also displayed part of the definitive script for jugs beneath the model itself, showing the current value of definitions — see Figure 4.23. Again, this display is created using dependency as much as possible. As a result, changes to the state of the jugs model, made by pressing the Fill/Empty/Pour buttons or by redefining the script, appear in the slide’s display of the definitive script. This was an attempt to make it easier for an audience to comprehend the indivisible linkage between the

```

sc(101, "Empirical Modelling for Graphics with tkeden", "titlef",
    "200");
sc(102, "Ashley Ward BEng MSc", "plainf", "s101y + s101h*2");
sc(103, "ashley@dc.s.warwick.ac.uk", "codef", "s102y + s102h");
sc(104, "Empirical Modelling Project Research Group", "plainf",
    "s103y + s103h*2");
sc(105, "Department of Computer Science", "plainf",
    "s104y + s104h");
sc(106, "University of Warwick, Coventry, UK", "plainf",
    "s105y + s105h");

%scout
display page1 = <s101win/s102win/s103win/s104win/s105win/s106win>;

```

Listing 4.11: EDEN script that instantiates the slide shown in Figure 4.22

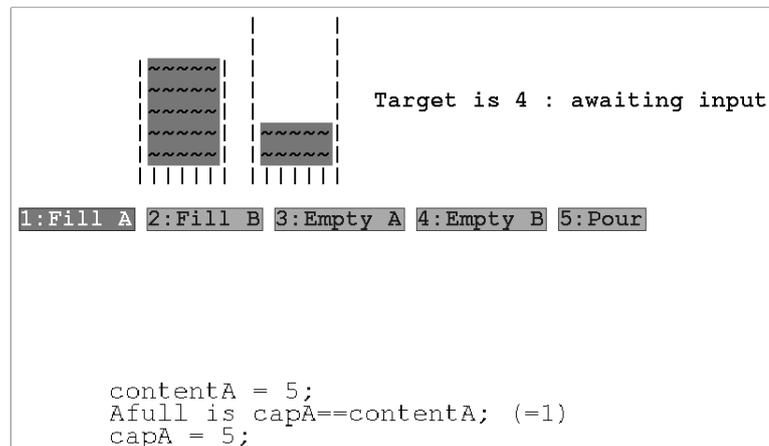


Figure 4.23: The 'jugs' model in a presentation slide

conventional-looking interface rendering of jugs and the unconventional, previously invisible, internal script state.

Models included on a slide in a presentation continue to ‘run’ when the slide is not visible, as EDEN maintains all current definitions in the system. Thought of in operating system (OS) terms, then, EDEN allows multiple models to be loaded²¹ and to “execute concurrently”. (Of course, EDEN does not provide many of the usual features of an OS and presently requires a conventional OS in which to function — but despite this, the analogy is interesting.) Unusually for an operating system, the data of executing ‘processes’ can be indivisibly coupled in un-preconceived ways. For example, the level of water in the left jug (`contentA`) can be linked to the *y* coordinate of the text on a particular slide, or to the size of a cube in the Rubik’s model (see earlier §4.2.1). The “inter-process communication” is handled by the EDEN definition maintainer.

4.2.7 Interface improvements

I have attempted to incrementally improve the `tkeden` interface at many points throughout the fifty versions. Figure 4.24 compares the interface from 1997 (when the tool ran only on one platform) to version 1.46 (October 2002). The menus available in the newer version are shown in Figure 4.25. In this figure, the new Edit and Help menus are the primary change from the 1997 interface.

Visible improvements in the user interface include new Help menus, containing condensed information on each notation. After observing many people make redefinitions in the wrong notation context, I added the radio buttons above the input window to make the current notation context more explicit and easier to change. To enhance the input window, I changed the background colour to white, improving readability both on screen and in printed screenshots, enlarged it slightly to show more context and improved the speed at which the cursor can be located by colouring it and making it flash. Other improvements include more keyboard shortcuts and more uniform use of common conventions — for example, using the ellipsis to mark items requiring further input in menus and showing the ‘Accept’ action as the usual action-implying button rather than choice-implying menu.

²¹Providing their variable identifiers do not clash — EDEN has only a global namespace.

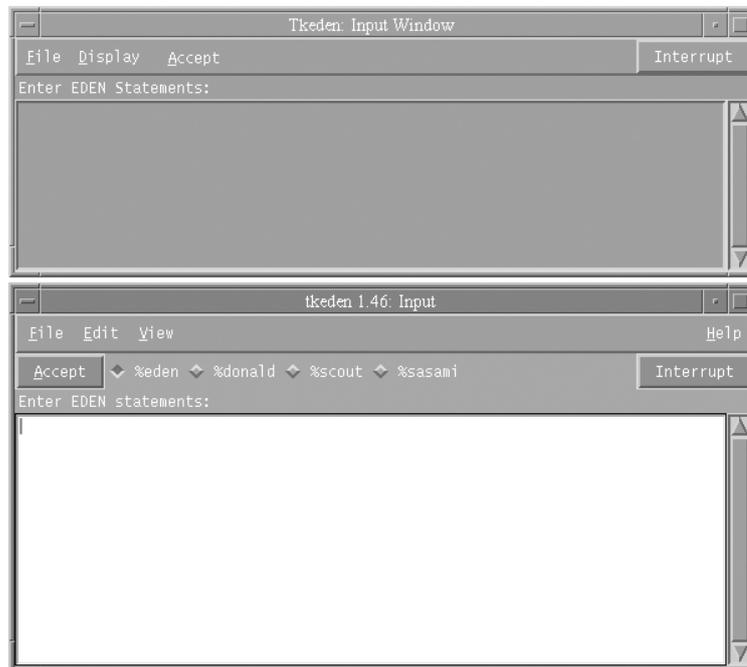


Figure 4.24: tkeden interface changes: tkeden-dec151997 contrasted with tkeden-1.46

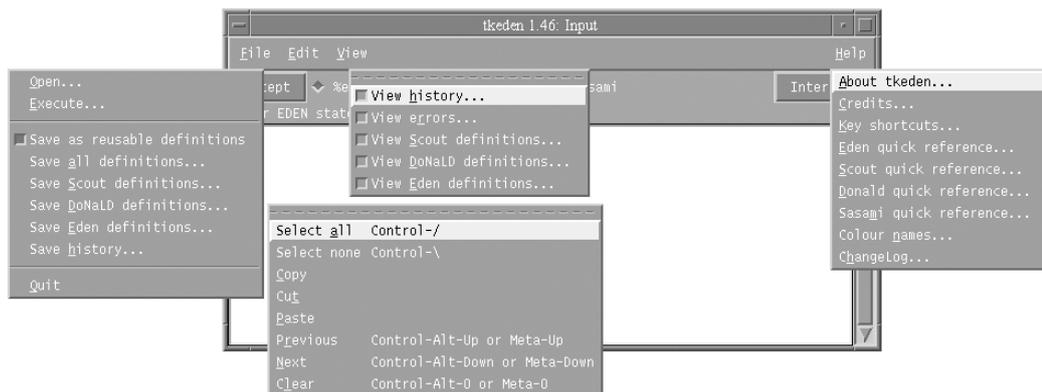


Figure 4.25: tkeden-1.46 menus

In addition, I improved the error messages. This work frequently went well beyond improving the text of each message, involving new implementation to present error information meaningful in the users' terms and the context for the error.

Although these improvements may seem trivial, where possible they are motivated by theory (e.g. the principles outlined in [app87]) and they do seem to have enhanced the usability of the tool and general acceptance to the audience — who are primarily undergraduate students using the tool for a relatively short amount of time. In addition, frequent releases of the tool have helped to generate feedback to steer the development. There is much scope for further improvement, particularly regarding input methods other than the input window — for example, editing the script in place, and mouse-driven creation of DoNaLD and SCOUT definitions. Wong [Won03, §7.3.2] includes an evaluation of the `tkeden` interface and includes further systematic treatment of interface issues in definitive environments.

One use of the presentation environment outlined in the previous section §4.2.6 shows a possible future direction for implementation of the EDEN interface. For a workshop presentation in 2003, Roe implemented a SCOUT model of a simplified version of the `tkeden` interface (which is currently implemented in Tcl/Tk), allowing integration of the interface into a presentation slide, as shown in Figure 4.26. Implementation of the entire interface itself in SCOUT (following the “in itself” theme) would allow use of dependency in implementing the interface, reduce the reliance on the present Tcl/Tk foundation and encourage tool users to modify the interface for their needs. A prerequisite here however would be various improvements to SCOUT, perhaps including the ability to use platform-native widgets.

The remainder of this section outlines another key aspect of improvements to the interface — the use of redefinition history.

Much of the potential attractiveness of a definitive tool comes from the ability to interactively make redefinitions to a model. If the interface does not support the modeller in easily and efficiently identifying and then making a series of redefinitions, then much of the power of the concept is in practice hidden from the user.

A key realisation that has guided some of my improvements to the EDEN interfaces is the common need (when undertaking exploratory modelling) to make a *series of similar* redefinitions. Often, feedback from making a redefinition leads to

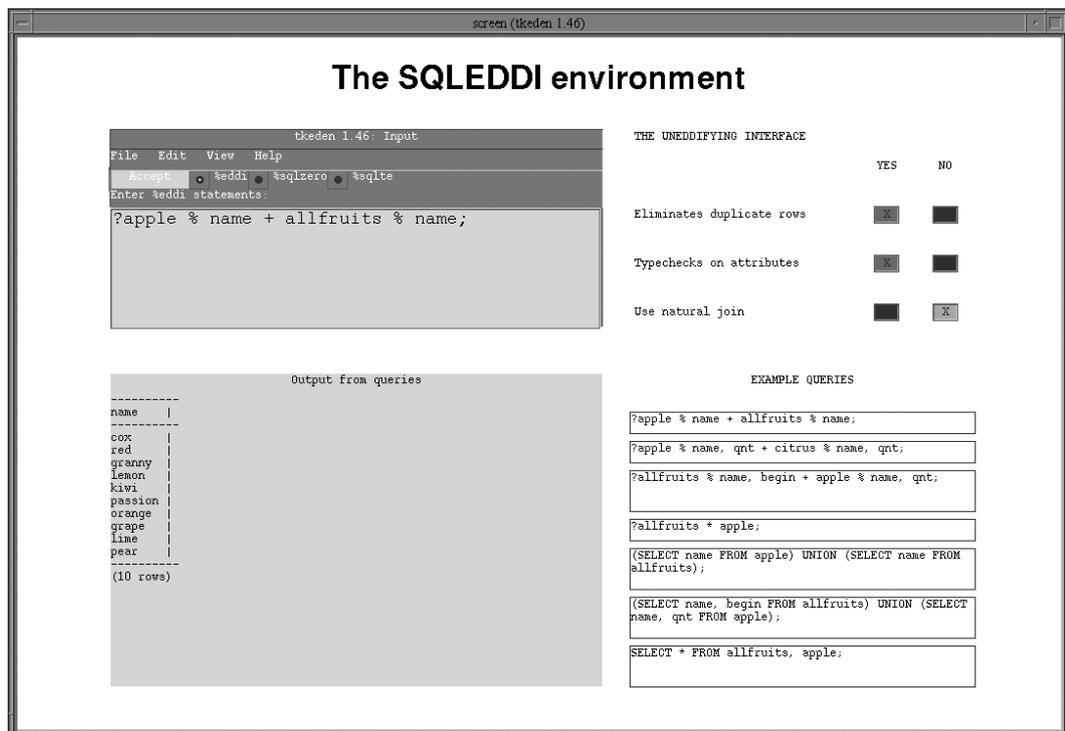


Figure 4.26: Part of a simplified tkeden interface constructed in SCOUT

the realisation that it can be improved in some way, and so it is desirable to be able to edit it. The original EDEN interfaces did not easily support this kind of activity.

In early versions of `ttyeden`, I took advantage of the GNU readline library [Ram] to implement cursor-driven line editing of the current input, including the ability to recall and edit previous input — the same facilities that are routinely provided in a UNIX shell. Later²² I implemented a similar facility in `tkeden`.

Users report that these additions have improved the usability of the interface greatly, and one hopes that the underlying concepts used in the tool are clearer as a result. However, these particular implementations are not quite satisfactory as they deal with only the textual user input and history thereof. There is a great potential for integration with the underlying machine state. First, a relatively trivial example is the ability to assist the user in making redefinitions, based on current state. The ability to automatically complete the typing of references to variables — similar to how in many UNIX shells, references to filenames can be completed by pressing the TAB key — would improve accuracy and speed of redefinition input.

More significantly, use of the *history* of the underlying machine state could lead to improvements that again would reveal more of the power of the definitive concept. It should in principle be possible to wind the machine state backwards, ‘undo’-ing through the history of the modelling session. Roe [Roe03, §2.3.1] reviews a similar facility in the Forms/3 tool [BAD⁺01].

The current tool saves the history sequence of interaction during a session to a file²³. The file also shows any error messages received, allowing the reader to distinguish successful and unsuccessful lines of input. I have made it possible to reinterpret this file without modification in most circumstances by introducing a new one-line comment syntax available in all notations with which to mark the errors, using the `##` character sequence to mark such a comment line. (The single `#` character could not be used for this purpose as it already denotes the list length operator in Eden.)

In his final year project [Kin04], Karl King has recently implemented a limited ‘checkpoint’ facility in `tkeden`, where the current state can be marked and later

²²In version 1.13.

²³Here, the sequencing of the lines in the file are usually significant, so pedantically this should not be called a ‘script’.

returned to. More generally, it should be possible to move backwards through the history sequence, and then move forward from a particular point, creating a history *tree* of redefinitions, the branches of which could then be later recombined in different ways. There are many precedents for this idea — for example, the RCS system [Tic85] and the later CVS system [cvs] allow ‘branches’ of source code history to be created and later merged, and the music production tool ‘Digital Performer’ allows branches to be created within audio editing history [mot]. Definitive state seems to be well-suited to such experimentation as referential transparency means the context needed for a definition is clear.

There are semantic problems with histories of redefinitions, however. For example: should the loading of a file of definitions be recorded in the history with a reference to the file identity or to the file contents? If the system makes a redefinition when the mouse is clicked, should this kind of redefinition be recorded in the history? Because of the conflation of tool and model, modelling and use, program and data in the history, there are no absolute answers to these questions — a problem which is exacerbated by an increasing amount of the tool being implemented “in itself”.

4.2.8 Novel interfacing

To conclude this overview of fifty versions of EDEN, I give three examples of recent novel usage of the tool, each involving interfacing EDEN to other systems through various extensions. The first two examples involve interaction with hardware devices, and the third interaction with external software. All three examples show various problems of concurrent synchronisation.

The Universal Serial Bus (USB) is an interface standard introduced in 1996 that allows connection of peripheral devices to personal computers. Devices are categorised into device classes which define generic behaviour and protocols. One such class is the Human Interface Device (HID) class. The USB and HID specifications are managed by an industry consortium and specifications can be downloaded from [usb]. Devices in the USB HID class include keyboards, mice, joysticks, data gloves, peripherals used for control of computer games (e.g. steering wheels, throttles, rudder pedals) and also devices that do not require human interaction but provide data

```

%eden

wheelfd = usbhidopen("/dev/input/event0");

proc readwheel {
    auto l;
    l = usbhidread(wheelfd);
    if (l != []) wheel = l;
    todo("readwheel()");
}

readwheel();

```

Listing 4.12: Using the USB HID facilities in `tkeden`

in a similar format to HID class devices, for example thermometers.

I introduced a capability for communication with USB HID devices in version 1.48 of `tkeden` on Linux, writing C code that builds on the support available on that platform to implement ‘built-in’ Eden routines `usbhidopen`, `usbhidread` and `usbhidclose`. The Eden code in Listing 4.12 uses this facility to continually read input from a USB steering wheel that is mapped to the Linux device `/dev/input/event0` and sets the Eden list variable `wheel` to the state read. The Eden `todo()` procedure is used to cause reading from the device to be interleaved with user input in a continuous loop.

After the code in Listing 4.12 has been introduced, the Eden variable `wheel` contains information read from the steering wheel. When the wheel is moved, the variable is changed by the `readwheel` procedure. The value can then be used definitively. The fourth item in the `wheel` list contains the angular position of the steering wheel. This can be connected to the position of the Sasami axes by introducing the following single Eden definition ‘`sasami_camera_rot is [0, 0, wheel[4]];`’. The introduction of this single definition causes the Sasami window to rotate in response to a move of the steering wheel, as shown in Figure 4.27.

Jon McHale, a third year undergraduate project student, built on this facility, constructing a `dtkeden` environment to be used for simulating reverse car parking (*carparkingsimMcHale2003*). The environment can be seen in use in Figure 4.28.

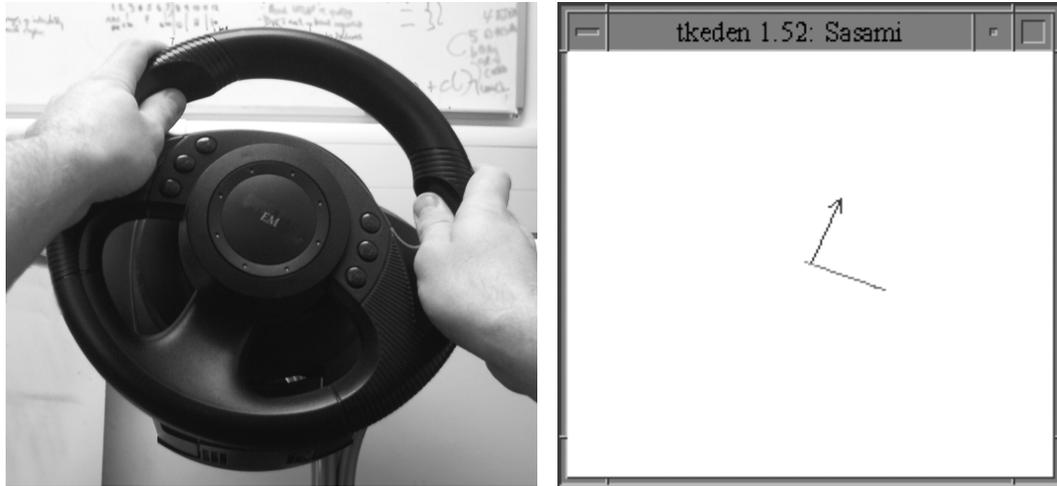


Figure 4.27: Sasami window linked to a USB steering wheel using a definition



Figure 4.28: Jon McHale and *carparkingsimMcHale2003*

McHale used four machines to display four views from the simulated car: front view, left and right wing mirrors and rear view (not shown in the figure). The simulation was constructed using the Eden, SCOUT, DoNaLD and Sasami notations — a SCOUT interface being used to configure an Eden model of the physics moving a two-dimensional DoNaLD rectangle ‘car’ within a 2D environment (the DoNaLD `intersects` function being used to check for collisions), which was linked by dependency to the 3D Sasami representations running on multiple workstations on a LAN — the distributed communication of redefinitions being a feature of `dtkeden`.

The second example of novel interfacing involves output to as well as input from hardware. Rod Moore and Stuart Valentine (department technicians) constructed a small OO-gauge railway based on the Leamington to Birmingham connection and a system based around a PIC16F877 microcontroller which controls the power applied to the points and the rails (the power to the rails is pulse-width modulated to give control of locomotive speed). The PIC also reads from a set of twenty reed switch sensors positioned around the track. Each reed switch is closed when a magnet attached to the locomotive passes by, allowing the location of the train to be determined at that moment in time. The PIC microcontroller communicates via a serial port, sending a packet of information containing the state of the set of reed switches when a change is detected. It can receive simple commands to apply power to the rails and change the state of the points through the same serial port. The system is shown in Figure 4.29.

Using techniques similar to those described above for USB HID, I added code to EDEN to allow communication with devices through an RS-232 serial port on Linux, adding `rawserialopen`, `rawserialread` and `rawserialclose` functions accessible from Eden. I then wrote Eden procedures as outlined in Listing 4.13 to communicate with the PIC microcontroller. These procedures cause redefinitions to the Eden variable ‘`trainMOTION`’ (for example, to the value “f7”, meaning “Forward” at speed “7”) to be propagated to the PIC microcontroller, and also the Eden variable ‘`sensors`’ to be redefined when the train passes a location sensor. The lower portion of the example in Listing 4.13 shows how, for example, the direction of travel of the train can be reversed whenever a signal is detected from a location sensor, demonstrating input and output.



Figure 4.29: Chris Rose and the model railway hardware

```
fd = rawserialopen("/dev/ttyS0");

proc sendTrainCommand : trainMOTION {
  rawwrite(fd, trainMOTION // "\r");
}

proc readFromTrain {
  c = rawserialread(fd, 1);
  /* omitted code: ... parse input read one character at a time
   from rawserialread. If input exists, redefine the
   'sensors' variable with the new state... */
  todo("readFromTrain()");
}

/* ... now, for example: */
direction = 1;

proc reverseOnPassingSensor : sensors {
  direction = !direction;
  trainMotion = direction ? "f7" : "r7";
}
```

Listing 4.13: Eden code to communicate with the train PIC

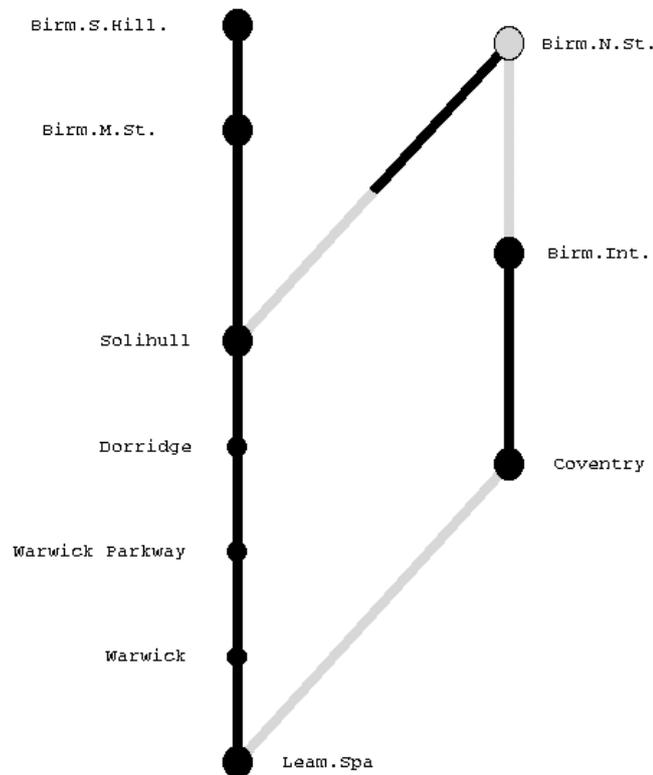


Figure 4.30: A partial screenshot of *modelrailwayRose2004*

Chris Rose, a third year undergraduate student, modified an existing `tkeden` model of a railway (*railwayYung1995*) to match the connectivity of the hardware model. The resulting model contained a schematic representation of the track layout constructed in DoNaLD, where the last received location of the train was indicated by colouring of a ‘train indicator’ node in the diagram — this being achieved by dependency on the `sensors` variable. The `tkeden` model (*modelrailwayRose2004*) is shown in Figure 4.30.

The original railway model (*railwayYung1995*) was a simulation whose progress was driven by a clocking procedure that is standard in many Eden simulation models. A simple version is below.

```
proc increaseClock : clock {
  todo("clock = clock + 1;");
}
```

Here, the `clock` variable is incremented as fast as the model can execute. (The semantics of the `todo()` procedure are further investigated in the next section §4.3.) The procedure can be modified to take account of real time, where the clock variable is not incremented until after (at least) the necessary amount of real time has passed. (I wrote a procedure to this effect for the car parking model, where it was desired to refresh the 3D graphical displays at a given real-time frequency.) In this type of ‘real-time’ solution, if the model re-evaluation takes a longer amount of time than there is available (a case which cannot be detected in the current EDEN), then the model will start to run more slowly than it should.

In the `tkeden` railway model that uses real hardware, progress of the `tkeden` model is driven by movement of the hardware, not a clocked simulation. The above `increaseClock` procedure is therefore replaced, and the `tkeden` model becomes in effect an extension of the hardware. Changes in the `tkeden` model are driven by changes in the hardware, and *vice versa*. There is no control flow or visible event loop in the `tkeden` model, just a set of definitions, some of which are dependent upon inputs from hardware. Interestingly, if the ‘train indicator’ definitions are present in the model, then they will respond to movement of the train, even whilst the remainder of the model is being interactively developed. This can be thought of as analogous to a prototyped section of circuit that is responding to inputs concurrently with another section of circuit that we are presently developing. In this analogy, stopping `tkeden` corresponds to removing power from the circuit: the current state is lost.

The final example in this section (before we generalise to some relevant issues from each of these examples) is motivated by the existence of the ‘pipeline’ notation translators mentioned in §4.1.4. Several translators, generally written in C with `lex/yacc` support, exist for definitive notations that are not integrated with the current EDEN system: for example, the ARCA to Eden translator written by Stuart Bird in 1991 (*arcaBird1991*). If it is desired to use these various notations, then the translator can be made to read a prepared file and write to a file that is then read into EDEN, but this is not a very satisfactory procedure when it is desired to interact with EDEN through the translator.

The generalised notations framework (see §4.2.4) would allow a notation managed by an external translator process to be used in the same way as other built-in notations (e.g. a radio button can be used to switch to and from that notation context), if Eden could be used to communicate with the external translator process. There are several technical difficulties here, however.

1. If the standard I/O library of the external translator detects that it is not connected to a terminal, it buffers input and output in blocks, as this makes file access more efficient. Therefore if it is connected to EDEN via a standard pipe, it cannot be used interactively.
2. The external translator may generate errors and these should be presented in the same way as other errors in EDEN.
3. Reading from the external translator cannot be allowed to block EDEN if there is no output from the translator.

I solved problem (1) by connecting a ‘pseudo-terminal’ between EDEN and the external process, following the example in [Cur96, Appendix D]. From the viewpoint of the standard I/O library of the external translator, it appears to be connected to a terminal and hence uses minimal buffering. Problem (2) was solved by connecting EDEN to the standard error output of the external translator using a pipe. If error output is detected in this pipe, the Eden `error()` procedure can be called, which causes the error to be handled in the same way as other EDEN errors. The necessary communication graph is shown in Figure 4.31. Problem (3) was solved by adding a function to EDEN wrapping the UNIX `select()` primitive, allowing a check to be made for input before a potentially blocking read is made.

I call the result an “Interactive Process Translator” facility. An external notation translator can be interactively introduced into the EDEN system by typing, for example:

```
%eden
installIPTrans("%arca", "/dcs/emp/empubliC/bin/arca.trans");
```

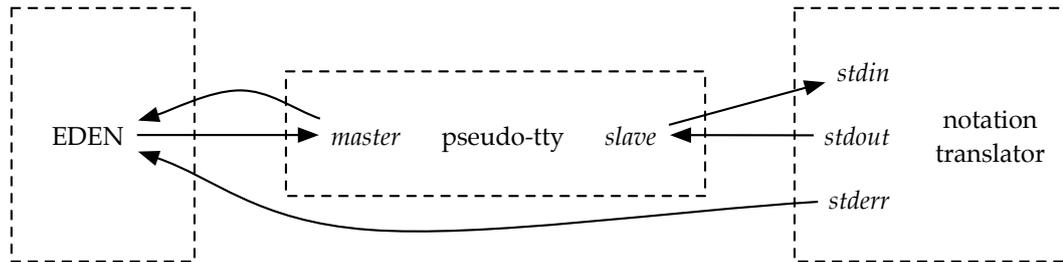


Figure 4.31: Communication between EDEN and an Interactive Process notation translator

Each of three above examples requires EDEN to read from a process or device which generates output asynchronously. The present solution in each case is to write an Eden procedure of the form:

```

proc readAsynchronous {
  if there is something to read {
    read the input
    deal with the input
  }
  todo("readAsynchronous()");
}

/* start the checking loop off... */
readAsynchronous();

```

The call to the Eden `todo()` procedure here causes the asynchronous input to be polled repeatedly. There are two problems. Firstly, if operating system mechanisms could be used by EDEN to cause the reading to be synchronised with the availability of some data to read, the inefficient polling would not be necessary. Secondly, the type of polling manifested by the `todo()` solution above is paused when EDEN is executing a sequence of statements, for example a `for` loop. User Eden code can therefore cause a delayed or missed response (depending on whether the input is buffered or not).

Both these problems emerge from the fact that the current EDEN is a sequentially executing machine, executing in only one operating system thread. The `todo()` procedure used here gives a partial solution, but a better solution requires concurrent execution within EDEN. The `todo()` procedure is investigated in more detail in the next section §4.3.

4.3 EDEN execution and scheduling

In practice, EDEN has proved to be by far the most effective ‘general purpose’ EM tool. It combines both dependency and agency in a way that we seem to find more acceptable than the ADM and the DAM machine, even making allowance for the relative immaturity of their implementations. Exactly how it works is therefore worth some investigation.

Y.W. Yung, who wrote the low-level machine algorithms that are still in use, starts his section on implementation [Yun90, §5] with a ‘rule of thumb’.

There are no strict rules for the behaviour of a definition maintainer. The evaluation of definitions can be eager, or lazy. The automatic re-evaluation can be enabled, or disabled, data driven or demand driven. No matter how the definition maintainer internally functions, it must ensure the values of formula variables are up-to-date. Perhaps it is the only rule for the definition manager to follow. I call it the “*Definition Manager’s Rule of Thumb*”.

Definition Manager’s Rule of Thumb: *Always evaluate a formula when a definition’s value is referenced (if the value is not up-to-date).*

Here, Y.W. Yung is stating evaluation/storage strategy 3 (evaluate-at-use-when-out-of-date), which sits somewhere between strategy 1 (evaluate-at-use) and strategy 2 (evaluate-at-redefinition). This strategy requires formulae, values and out-of-date flags to be stored — it uses the maximum storage of the three strategies, with the aim of minimising the amount of evaluation. It is the most complex of the three strategies to implement and describe, which is why this chapter on EDEN appears after those on ADM and DAM.

Y.W. Yung does not state how the current EDEN maintainer operates. He does give an algorithm, but this describes the same depth-first evaluation scheme given in his original project report [Yun87], which was used only in very early versions of EDEN for which the source code is no longer electronically available²⁴. The current EDEN uses a breadth-first evaluation scheme, about which Y.W. Yung gives only an example trace and a few short paragraphs [Yun90, §5.5.2]. The aim of this section is to describe the algorithm used and its consequences. This will be done by examining the artefact that remains — the tool — firstly as a black box, and secondly by examining the source code.

²⁴Although a printed copy exists in Y.W. Yung’s project report.

4.3.1 Terminology mostly explicitly dismissed

We have already briefly described the difference between Eden's formula variables and actions in §4.1.2. Now that we are examining the implementation more closely, it is necessary to look more closely at the distinctions made.

Y.W. Yung [Yun90, pp.4–5] makes a distinction between 'implicit' and 'explicit' actions:

A spreadsheet is a program that provides us with a large grid of cells into which we can insert numbers and formulae. A cell that contains a formula causes the system to re-calculate the formula whenever any cell on which it depends is changed, and display the new value on the screen automatically. The display action is implicitly invoked by the system after the re-calculation. These actions can be called *implicit actions* because they are pre-defined in the system.

The `make` command in UNIX is a good example of file maintenance software. Unlike the spreadsheet software, the user has to specify the updating action in a file (the `makefile`) explicitly. These actions can be called *explicit actions*.

This terminology is then used in the statement of the primary goals for the Eden language [Yun90, p.26]:

1. to support general purpose programming (e.g. in order to implement definitive notations)
2. to support definitions
3. to support explicit actions (since they are extensible and much more powerful than the implicit actions, they fulfil the general purpose requirement).

Y.P. Yung [Yun93, p.92] follows this distinction, repeating the example of display-updating actions in a spreadsheet and again naming this 'implicit' action:

We can imagine that there are implicit actions which update the values of the variables on the screen. Similarly, there are such implicit actions in the implementations of definitive notations. For example, each graphical object in DoNaLD has a representation on the screen. For a `point` variable there is a dot to represent it; for a `line` variable, there is a linear set of points and so on. So there will be a `plot_point` action in the implementation of DoNaLD to be called automatically when a `point` variable is updated, and likewise a `plot_line` action for a `line` variable.

However, the terms 'implicit' and 'explicit' have been used in different senses since. The implicit/explicit distinction in *action* is in fact the reverse of a distinction sometimes made between implicit and explicit *dependency*. This distinction is commonly made in sources on object-oriented programming. Cartwright, in the overview documentation for his JaM2 API (jam2Cartwright2001/JaM2/overview.html) which

follows on from his thesis work [Car99], defines the terminology in the following way:

Two distinct forms of dependency exist, explicit and implicit:

- explicit — Dependency is expressed in a well-defined underlying algebra between values of simple data types such as numbers and strings of characters, using well-defined functions over those data types. An actual spreadsheet is an example of explicit dependencies (although the way they are maintained internally by the spreadsheet executable may not be).
- implicit — Dependency is expressed through a system of messages sent to software objects (class instances or a pointer to a value of an abstract data type) to signal when they should update. This process relies on code inside procedures or methods performing the correct update. The `make` application that is used to maintain dependency between source files for an executable when it needs to be rebuilt is an example of this.

Heron [Her02, pp.16–17] clarifies this further:

Explicit dependency is that found within a spreadsheet and within definitive scripts, it relies on a formal notation. The spreadsheet definitive machine can interpret and work out the dependencies from the script itself. Implicit dependency as found in a `Makefile` relies on side effects caused by procedural commands (i.e. compiling). Since there is no formal notation to describe these side effects then the dependency maintainer (in this case `make`) requires that the dependencies are spelt out to it.

Traditional programming makes no use of explicit dependency and any dependencies that exist (whether they are message passing, propagating updates, etc) require specific procedural actions to be written and maintained by the programmer to keep the dependencies valid.

... The most successful and widely used Empirical Modelling tools have combined the powerful explicit dependency of definitive scripts with the less formally defined implicit dependency of procedural programming. Tools such as EDEN combine definitive notations with functions and procedures in a similar way to the modern programmable spreadsheet.

In Eden, therefore, explicit dependency is maintained by implicit actions. Explicit actions may be used to maintain implicit dependency.

Authors therefore vary in their meaning of the terms implicit and explicit. Sources (including, but not limited to the above) variously emphasise the following:

- Formality of notation (hence, analysability) used to describe when the action should occur;
- Instigator of the action: the system or the user?
- Specifier of the action: the tool implementer or the tool user?

- The ‘defined-ness’ of the references used: are literal values stated or references to other values?

In the sections below, the words implicit and explicit are (explicitly!) avoided. It is necessary however to build on the terms initially defined in Appendix §3.A (p.178). The sections below extend the use of the terms below from the script graph context to Eden definitions and actions in the following way:

Triggers of change are what *cause* change;

Targets of change are what the change will *affect*;

Sources are what an action *observes*.

The terms sources and triggers are synonyms where definitions are concerned, since definitions observe only their triggers, as will become clear.

4.3.2 Requirements of an EDEN-like definition maintainer

With the benefit of examining the ADM and the DAM machine, I can now state requirements for an EDEN-like definition maintainer (DM) that are firmer than Y.W. Yung’s rule of thumb.

When redefinition(s) are made, a definition maintainer must re-evaluate the necessary atoms and arrive at a final steady state. In general, the redefinitions can be a *set* of changes. The BRA is explicitly designed to process a set of redefinitions. EDEN has an ‘autocalc’ variable that can be used to form a set of redefinitions.

In Figure 4.32, I state three types of requirements for an EDEN-like DM. The requirements are necessary for correctness, necessary for efficiency or may be desirable respectively.

Necessary requirements

1. In the final stable state, (at least) all target atoms of the change set must have been re-evaluated.
2. In the final stable state, each atom must have been re-evaluated *after* its source atoms.

Requirements for efficient performance

3. Only atoms that are targets of the change set need be evaluated. (This is a corollary of requirement 1.)
4. An atom should not be re-evaluated more than once during a single transition.

Possibly desirable features

5. We may want to separate definitions and actions, evaluating definitions before the actions so that the actions observe consistent definitive state. Alternatively this can be thought of as the definitions taking priority over the actions. (Y.W. Yung suggests this as a way of separating the ‘mathematical’ and ‘feedback’ parts of a model.)
6. We may wish to minimise the total memory used. A recursive solution requires more stack memory as it uses a nested calling mechanism.
7. We may wish to minimise the amount of computation performed. A non-recursive solution will have a time-consuming sorting phase.
8. We may want to detect and prevent cyclic dependency.

(Note: the ‘sources’ and ‘targets’ referred to above are determined from the script graph *after* the set of redefinitions have taken effect.)

Figure 4.32: Requirements of an EDEN-like definition maintainer

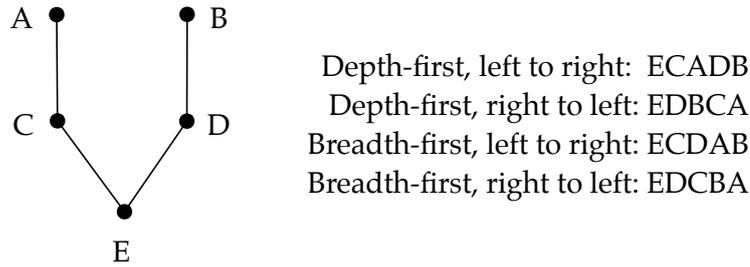


Figure 4.33: Depth- and breadth- first topological sorts

Requirements 2 and 4 in Figure 4.32 can be achieved by a topological sort, which is an operation that embeds the partial ordering described by a script graph (see Appendix §3.A, p.178) into a linear ordering which can then be used as an evaluation sequence. Two ways of implementing a topological sort (*cf.* §3.1.2) of the nodes of a script graph are depth-first and breadth-first, which are usually implemented recursively and iteratively respectively. To fully determine the ordering, it is also necessary to know the order in which the edges emanating from each node are processed. A basis for this ordering could be established on a geometric basis with reference to a figure (e.g. leftmost to rightmost) or on a temporal basis (e.g. oldest to newest). Figure 4.33 shows four different topological sorts for a particular graph.

4.3.3 Black-box analysis of EDEN

This section gives an analysis of the current EDEN DM by systematically constructing various definitive scripts and observing how they are re-evaluated. The approach taken here could be used to evaluate any other DM.

Firstly, scripts containing only definitions are investigated. Definitions in Eden can use any user-defined function. It is possible to write a user-defined function which includes side-effect(s). This means it is technically no longer a function, but if we only print literally defined state, this will not affect the DM scheduling.

The scripts shown in Figure 4.34 therefore use a user-defined function to print a trace when evaluation occurs, making evaluation visible. The results are explained below. They have been obtained by running the scripts in `ttyeden-1.46`.

Script 1 is a fully constrained evaluation ordering. When ‘**d**’ is changed (denoted in the diagram by the bold font), the only possible correct way to evaluate script 1 is ‘c’, ‘b’, ‘a’. EDEN evaluates this script in this way.

Script 2A in Figure 4.34 reveals a choice of breadth- or depth-first evaluation orderings, as illustrated in Figure 4.33. EDEN evaluates this script using a breadth-first ordering. Script 2B shows that EDEN is using a temporal basis for edge ordering: when there is a set of atoms with no ordering constraint mandated by the script graph, EDEN evaluates them in the order of their (re)definition, oldest to newest.

Script 3 could lead to double evaluation in some DM implementations. EDEN however evaluates the definition ‘a’ last, complying with the necessary requirement (2) from Figure 4.32, and evaluates it only once, complying with the efficiency requirement (4).

Scripts 4A, 4B and 4C show the results of making changes to a *set* of definitions. In each example, the multiple changes could lead to multiple unnecessary evaluations. In Eden, a set (or, in Cartwright’s terms, a block) of redefinitions is formed using the `autocalc` variable. Script 4C (which also appears as the ‘power’ example in Figure 3.2, p.115 *et seq.*) is taken from Cartwright’s thesis. Contrary to Cartwright’s assertions about `tkeden` (quoted in §3.1.2) that EDEN “...often performs a large number of unnecessary calculations”, EDEN in fact optimises re-evaluation across the set of changes. This result shows that the BRA has no particular advantage over the algorithms used in EDEN (other than that the BRA is well documented by Cartwright — itself a big advantage). This finding reinforces the conclusions reached on the basis of performance measurements comparing !Donald and EDEN in §3.4.4.

Continuing systematically, we now investigate scripts containing only actions, in order to examine the evaluation strategies employed by EDEN here. We shall reuse the examples just presented, translating the definitions into actions.

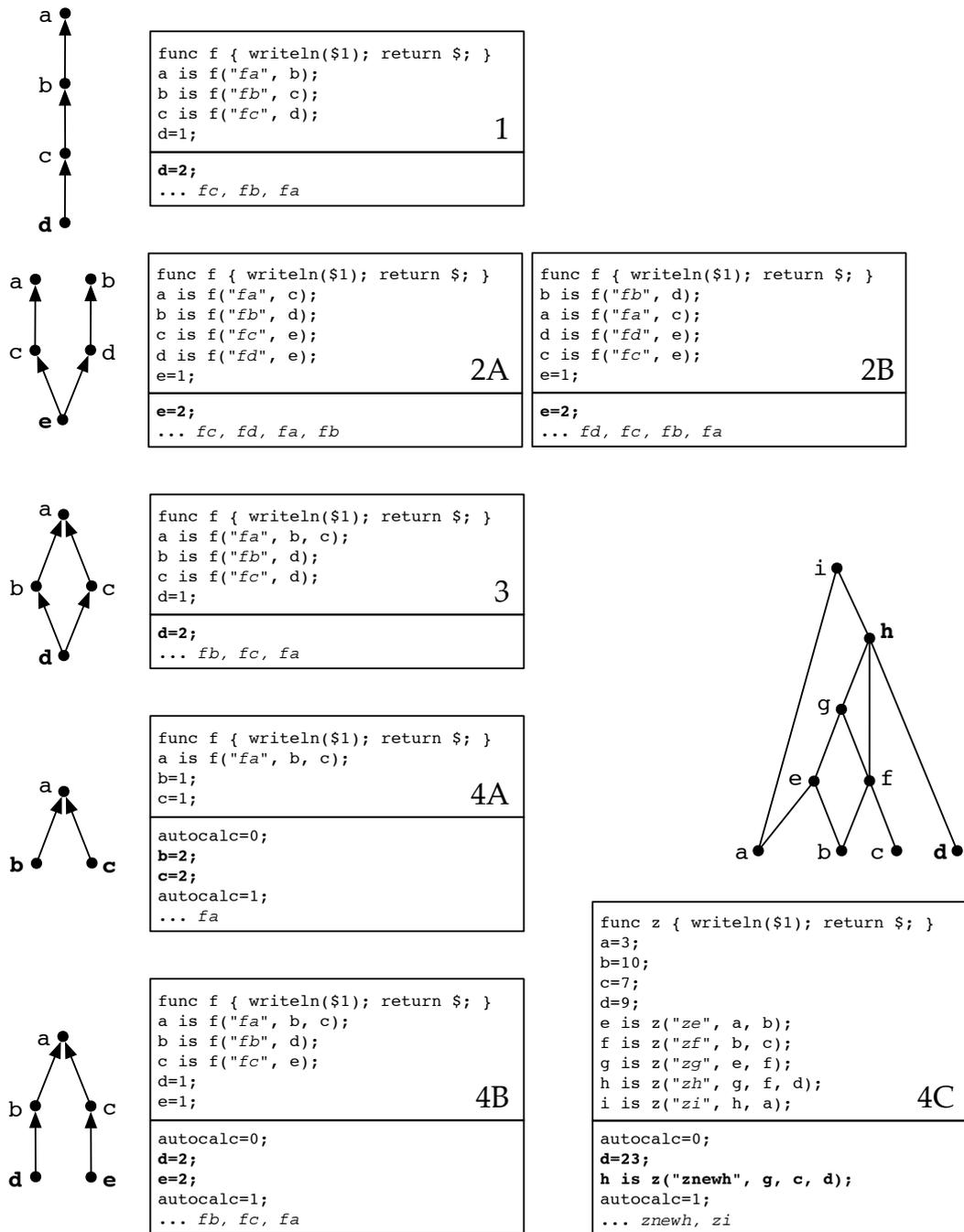


Figure 4.34: Definitive script graphs containing only definitions systematically constructed to examine DM evaluation strategies

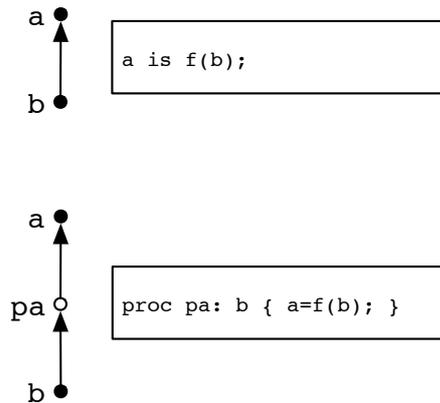


Figure 4.35: An equivalent definition and action

In Eden, the value of an action is the action declaration itself. The actions therefore cannot represent the values of the definitions directly. Translation from definitions to actions therefore involves the introduction of more atoms: the actions themselves. The introduction of the extra atoms is the topic of §5.1.1. Here, we make the following transformation. The definition:

```
a is f(b);
```

can be transformed to the action:

```
proc pa: b { a = f(b); }
```

The diagrammatic conventions used here for this transformation are shown in Figure 4.35 (which, like Figure 4.34, does not show the function f).

The examples in Figure 4.36 show that when scripts containing only definitions are translated to only actions using the transformation scheme shown above, then the actions are evaluated in the same way as the original definitions were. This is what Y.W. Yung means by the ‘equivalence’ of definitions and actions.

It may seem strange that EDEN manages to order the evaluation of the actions as well as the definitions, despite the DM lacking knowledge about what actions actually write to. We shall see that the EDEN scheduler actually schedules definitions and actions in the same way, and in a way that does not require full knowledge of the consequences of an action.

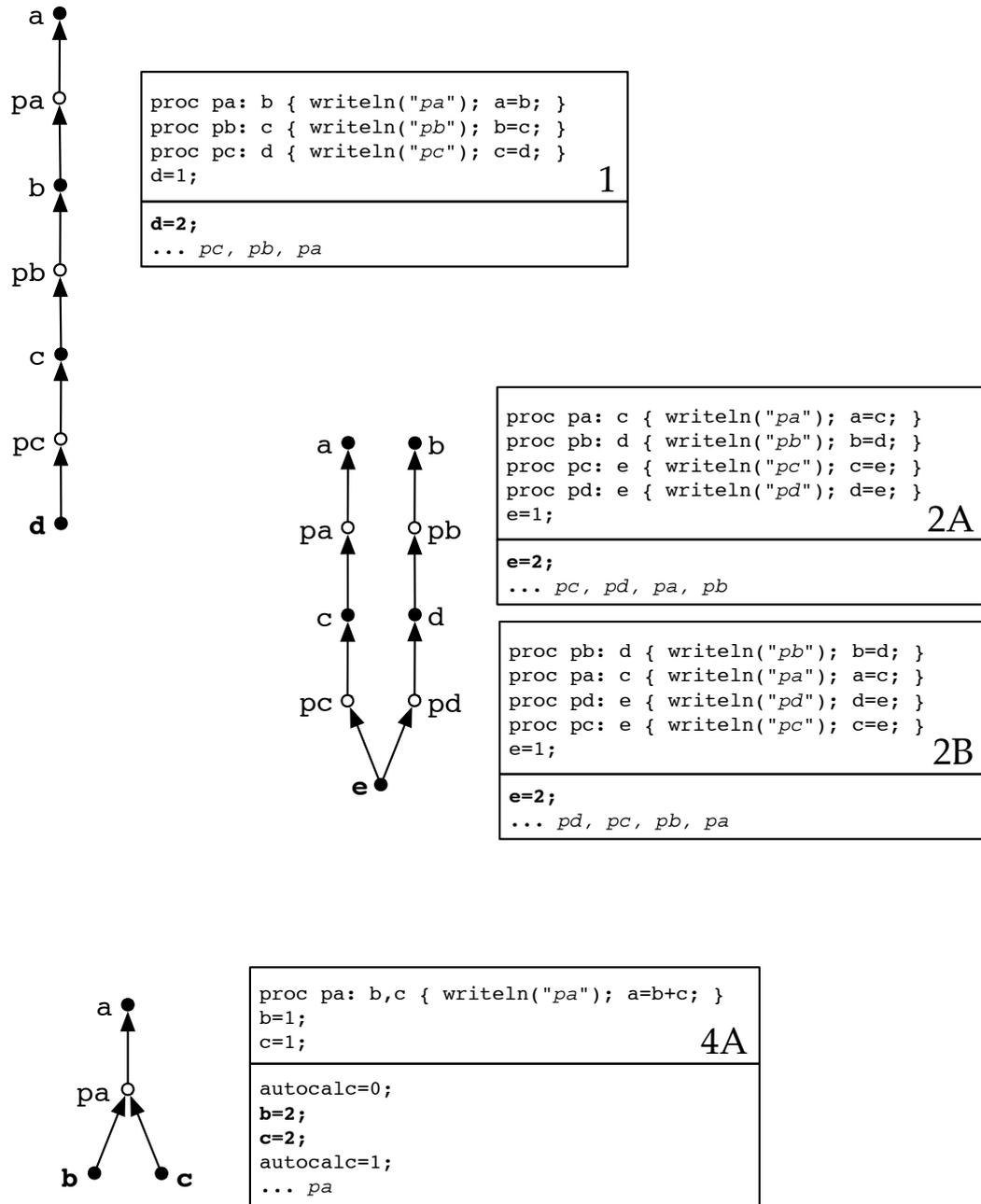


Figure 4.36: Definitive script graphs containing only actions systematically transformed from Figure 4.34, which contains only definitions

4.3.4 Differences between definitions and actions in Eden

Having examined the ‘equivalence’ of definitions and actions under the described transformation scheme, we are now in a position to elaborate the differences between definitions and actions more clearly. The limits of the transformation scheme used show that actions can do more than definitions. This gives us insights about *both* definitions and actions.

Constraints on definitions in Eden

- A. A definition writes only to a single well defined atom, stated on the LHS of the ‘is’ (the target).
- B. A definition is the only agent in the system that writes to the atom specified on the LHS of the ‘is’ (the target).
- C. A definition reads only from atoms stated on the RHS of the ‘is’ (the sources/triggers).

Freedom of actions in Eden

- D. An action can write to any atom, including something stated on the RHS of the colon (the triggers). (Compare (A).)
- E. An action can write to any *set* of atoms. (Compare (A).)
- F. It is not possible to tell what an action writes to without executing it. (Actually, it might be possible in some limited cases, but the current EDEN makes no attempt to tell.) (Compare (A).)
- G. It is possible for a single atom to be written to by more than one action. (Compare (B).)
- H. An action can read from an atom that is not stated on the RHS of the colon (a trigger). (Compare (C).)

4.3.5 Inside the machine: definitions and actions

Looking at the source code, the core of the EDEN machine is actually a three line segment of C code, which was a part of the original hoc implementation (by Kernighan and Pike — see §4.1.1). It is shown below, slightly simplified.

```
while (*pc) {  
    (*(pc++)) ();  
}
```

In informal English, this reads:

Whilst the function pointer (or virtual opcode) at location `pc` is not 0:
 Call the function pointed to (involving a double dereference),
 Increment `pc` to point to the next function pointer in the array.

This core is a virtual machine which executes an instruction stream of function pointers, rather as the Java Virtual Machine executes bytecodes.

Definitions and actions are both encoded into streams of instructions which are executed in order to enact state change when the scheduler deems it necessary. The execution mechanism is the same for both definitions and actions. We can therefore think of definitions as being maintained by *definition agents*.

What then, is the technical distinction between definition and action? There are two parts to the answer. The first part has been given in the previous section. Points A to C represent *constraints on definition agent action*. The second part of the answer involves priority in scheduling. Roughly speaking, definitions take priority over actions so that definitive state always appears consistent. The full detailed picture is given in the next section.

Restrictions on reading and writing to state based on agent identity have long been at the centre of abstract data type and object-oriented thinking. Definitive systems however use the script graph as a basis for restrictions rather than modules, and the restrictions are combined with evaluation scheduling and prohibition of graph cycles.

4.3.6 The EDEN scheduler algorithm

Although largely explained without reference to the source code, the insights in the previous sections have been derived with the help of detailed inspection of the code (which as Figure 4.9 on p.219 shows, is currently 30,000 lines of C).

I have reduced the source code representing the scheduler algorithm (which is spread across five source files in the present code) down to the pseudo-code shown

in Appendix 4.A on p.277. I have renamed some of the functions for clarity — the renaming is documented in the box numbered 5.

From the description of the algorithm, I have produced the diagram showing the EDEN machine data, operations, data flow and control flow shown in Figure 4.37. This diagram reveals part of the difficulty of analysing the code: many operations lead to the `execute` procedure at the base of the figure, corresponding to the three line virtual machine implementation described in the previous section. The virtual machine can execute any function pointer, and here one of the difficulties arises — many of the function pointers correspond to a re-entrant call into the machine, shown as a line looping from the bottom to the top of the figure. Still, the diagram is helpful for much analysis about the EDEN scheduler.

The final reduction I have made to describe the EDEN scheduler is shown in Figure 4.38. This set of diagrams is an attempt to represent the operational contexts of the EDEN machine more statically than in Figure 4.37. It clearly shows the dual nature of the EDEN machine, which evaluates both speculatively²⁵ and on demand. Demand-driven evaluation occurs when the statement being executed reads from definitive state that is marked as `OutOfDate`. Speculative evaluation is initiated by change, and can be inhibited by changing the setting of `autocalc`. This is how sets of redefinitions are processed efficiently: when speculative evaluation is inhibited, changes still cause scheduling of necessary definitions and actions, but the evaluation (diagram 3 in Figure 4.38) is not actually performed.

Another major insight contained in these representations of the EDEN scheduler is that EDEN can be viewed as a stack of virtual machines, each machine implementing an indivisible view of state for the machine above. There are presently three such virtual machines: definition processing is at the bottom, actions execute over definitions, and an interface automation layer executes over actions. Thus, a triggered action observing definitive state will always observe it in the `UpToDate` state²⁶. We have seen that the scheduling of definitions and actions is very similar — but the scheduling operates on two different levels. One layer up, the interface will only display states that occur after processing of actions has completed — stated

²⁵On the assumption that an `UpToDate` variable will be observed before it is changed again.

²⁶Although there is a caveat here to do with `adjacentSourcesAreUpToDate` of the definitive state.

4.3.6. The EDEN scheduler algorithm

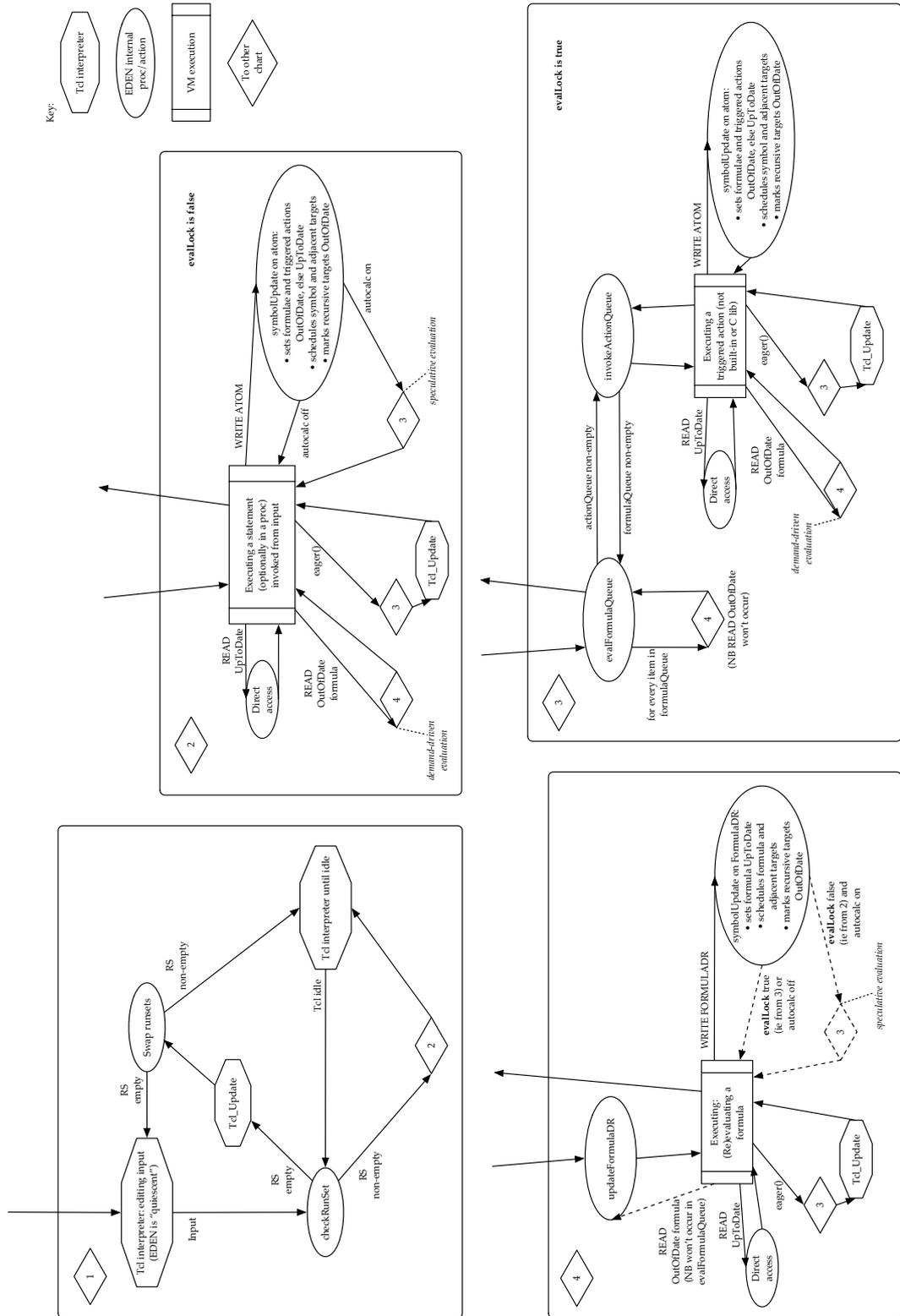


Figure 4.38: The EDEN machine states considered more statically

conversely, the interface will never show the state mid-action²⁷. This structure is similar to that suggested by Dijkstra in his “Notes on Structured Programming”, where he expresses his wish for meaningful state when a virtual machine is between instructions. He states [DDH72, pp.49–50]:

... we have arranged our program in layers. Each program layer is to be understood all by itself, under the assumption of a suitable machine to execute it, while the function of each layer is to simulate the machine that is assumed to be available on the level immediately above it.

...

We may hope that the model will give us a better grip on the problems that arise when a program has to be modified while it is in action. If a machine at a given level is stopped between two of its instructions, all lower machines are completely passive and can be replaced, while all higher machines must be regarded as engaged in the middle of an instruction: their state must be considered as being in transition. In a [conventional] sequential machine the state can only be interpreted in between instruction executions and the picture of this hierarchy of machines, each having its own instruction counter — “counting its instructions” — seems more profitable if we wish to decide at any given moment, what interpretations are valid. In the usual programming language in which computational progress is measured in a homogeneous measure — say “the grain” of one statement — I feel somewhat helpless when faced with the question of which interpretations are valid when.

My current best conceptual model of EDEN execution follows this notion of layers of indivisibility. In this model, an action making a change to a variable value can cause the definition layer to gain priority and bring the definitive state back into consistency, followed by resumption of the previous action at the previous level. A transition of priority in the reverse direction can also occur if a change is propagated via dependency to a value which is a trigger for an action. Transitions of priority in this direction however do not take place immediately — actions are queued until the definition layer has finished executing. This conceptual model is a helpful guide (and it also makes the sequential nature of EDEN clear), but it is unlikely to capture all of the actual behaviour of the current scheduling algorithm.

Unfortunately the actual layers (such as they are) are only made visible in the current implementation when diagrams such as Figure 4.38 are drawn. The notion of layers may be helpful in planning a new implementation, however. Y.P. Yung in fact makes such a proposal in [Yun96] which will be examined in the next section.

²⁷There is a caveat here too, as the `eager()` command can be used to force the current state to be propagated to the interface.

4.3.7 Higher Order Definitions (HODs) in EDEN

This section addresses the question: does EDEN need another layer in order to cope with HODs?

Y.P. Yung proposes [Yun96, p.27] but does not fully explain a “4-layer prioritised action system” in an attempt to rationalise the multiple action control sub-systems in EDEN. The proposal comprises a ‘definition layer’, ‘meta-definition layer’, ‘decision layer’ and ‘transition layer’. The definition layer maintains the state specified by definitions. In this layer, multiple triggers may be safely eliminated. The meta-definition layer maintains Higher Order Definitions²⁸ where “because the aim is still to maintain the consistence of state, hopefully we can eliminate multiple triggers”. The decision layer is needed “to determine what actions are needed to transit the state... Actions in the Decision layer may still interfere. However it becomes apparent that this kind of interference is the same class as the interferences between concurrent agents that we are more comfortable to live with.” Finally, the transition layer actually performs the actions determined by the decision layer.

I did initially believe that a HOD layer was necessary, but I thought that it would need to be added at the bottom, since HODs change the script graph, which is assumed to be static by the definition layer. After having analysed the EDEN scheduler at the level shown in the previous section, however, I no longer believe such a layer to be necessary. The key realisation here is that the EDEN scheduler only schedules adjacent targets of any change (although it marks *all* targets recursively as OutOfDate). As it does not construct a deep schedule, it is flexible enough to cope with actions changing the script graph.

A standard example in this thesis is the ‘if’ HOD:

```
v is b ? x : y;
```

(in English, if **b** is true, **v** is **x**, otherwise **v** is **y**). The problem with the definition as stated is that **v** is dependent upon both **x** and **y**, whereas the dependency should be upon *either* **x** or **y**, the choice itself dependent upon the current value of **v**. The following example illustrates the problem.

²⁸HOD is the term used by [Yun90, p.103] to describe “a definition of a set of definitions”. Y.P. Yung [Yun93, p.115] calls these “meta-definitions (structures that generate definitions)”, but the term HOD has continued in usage since.

```
v is b ? x : y;
b is 1; x is 2; y is 3;
proc p : v { writeln("v has changed"); }
x is 4; /* p is triggered */
y is 5; /* ! p is triggered, incorrectly */
```

With the added confidence in EDEN's scheduler, however, we can write the following:

```
proc cv: b { if (b) { v is x; } else { v is y; } }
```

This action reads very much as the English description. It redefines `v` when `b` is changed. Such Eden code has not seemed popular in the past — and there were in fact some bugs, initially discovered by Carlos Fischer (a visiting researcher), relating to triggered actions that make redefinitions, which I fixed in version 1.13. Perhaps we can now start writing HODs in this style.

4.A Pseudo-code of the EDEN scheduler algorithm

1	<pre> RUNSET ----- User input from Tk Input Window, todo() input call... queue(cmd) add cmd to the tail of the RS list Tcl_DoWhenIdle(checkRunSet) checkRunSet() if (interrupted) clear run sets if (RS is not empty) while (RS is not empty) get an action string, s, from the head of the RS list run(s) Tcl_DoWhenIdle(checkRunSet) else Tcl_Update swap runsets if (RS is non-empty) Tcl_DoWhenIdle(checkRunSet) PARSER ----- execute(), include() and many other routines that need to invoke the parser on a string or a file call... run(s) Invoke the parser on string s to generate VM code p execute(p) </pre>
2	<pre> CYCLICITY CHECK ----- 'is' redefinition, '-->' related-by declaration call... cyclicCheck(sp, sources) Recursively search the sources list for an occurrence of sp. Mark the sources encountered along the way in order to determine when the job is done. When the job is done, search again and remove the markers. The Eden function dcc() can be used to globally disable these checks. TRIGGER ----- '=' 'l1/g' '+' '=' '-' assignment, '++' '--' post/pre increment/decrement, 'shift', 'append', 'insert', 'delete' list procedural ops, 'func/proc/procmacro' declaration, 'func/proc/procmacro' triggered action declaration, 'is' redefinition, '-->' related-by declaration, ... all VMOPs call... symbolUpdate(sp) if (sp is a newly redefined triggered action or definition, defined by the caller) mark sp OutOfDate else mark sp UpToDate schedule(sp) scheduleAdjacentTargets(sp) if (autocalc) evalFormulaQueue procmacros, use of Eden eager() procedure call... eager() VMOP save the value of evalLock evalLock = FALSE evalFormulaQueue restore the value of evalLock Tcl_Update touch() for each argument, sp, provided to touch() scheduleAdjacentTargets(sp) evalFormulaQueue pushUNDEF </pre>

4

```

invokeActionQueue()
if (evalLock is TRUE)
  return
evalLock = TRUE
while (actionQueue is not empty)
  remove the front item, sp from actionQueue
  if (adjacentSourcesAreUpToDate(sp))
    call(sp, no arguments) // this may add items to the formula and
    // action queues
  discard the value returned
evalLock = FALSE
// the actionQueue is now empty
if (the formulaQueue is not empty)
  evalFormulaQueue

lvalue evaluation, l=l//q optimisation, back-ticks
check for an OutOfDate formula and sometimes autocalc
and call...

updateFormulaDR(sp)
if (UPDATE) -- used to prevent evaluation by the ? query operator
  execute(VM code associated with sp)
  store the value returned in the data register associated with sp
  symbolUpdate(sp)

func/proc/procmacro invocation
call...

call(sp, arguments)
save information about the current frame
if (sp is a proc, func or procmacro)
  push the arguments
  push UNDEFs for local variables
  execute(VM code associated with sp)
else if (sp is a built-in or c library function)
  push the arguments
  appropriately invoke code associated with sp
  restore frame information
  leave the item returned on the stack

eval(expr) calls execute on expr before storing the result,
back-ticks
call...

execute(p)
  execute VM function pointer code at p,
  until a rts (zero) VM code is found
  ... NB VM code can include any VMOPs

```

3

```

SCHEDULE -----
schedule(sp)
if (sp is a formula)
  Q = formulaQueue
else if (sp is a proc, func, procmacro or built-in)
  Q = actionQueue
else
  return
if (sp is already in Q)
  move sp to the end of Q
else
  if (sp is a formula)
    append sp to the end of Q
  else if (sp is a proc, func, procmacro or built-in)
    if (sp has sources) -- ie it is an action
      append sp to the end of Q

scheduleAdjacentTargets(sp)
for each adjacent target, t of sp
  markRecursiveTargets(t)
  schedule(t)

markRecursiveTargets(sp)
if (sp is UpToDate)
  mark sp as OutOfDate
for each adjacent target, t of sp
  markRecursiveTargets(t)

EVALUATE -----
evalFormulaQueue()
if (evalLock is TRUE)
  return
evalLock = TRUE
while (formulaQueue is not empty)
  remove the front item, sp from formulaQueue
  if (sp is OutOfDate)
    if (adjacentSourcesAreUpToDate(sp))
      updateFormulaDR(sp) // this may add items to the formula and
      // action queues
evalLock = FALSE
// the formulaQueue is now empty

if (the actionQueue is not empty)
  invokeActionQueue
// the actionQueue and formulaQueue are now both empty

```

5

NOTES

- Some procedure arguments have been simplified
- `sp` is `OutOfDate` is when `sp->changed=TRUE`,
 `UpToDate` when `sp->changed=FALSE`
- This pseudo code omits mention of the "master" stack of agent names
- `setprompt()` has been omitted from `checkrunset()`
- `tcl_Update` is actually `tcl_EvalEC(interp, "update")`
- `symbolUpdate` simplifies setting of `OutOfDate` flag by caller

HERE NAMED	ACTUAL NAME	FILE LOCATION
<code>formulaQueue</code>	<code>formula_queue</code>	[main.c]
<code>actionQueue</code>	<code>action_queue</code>	[main.c]
<code>evalLock</code>	<code>lock</code>	[refer.c]
	<code>queue</code>	[eval.c]
<code>cyclicCheck</code>	<code>checkrunset</code>	[machine.c]
<code>symbolUpdate</code>	<code>checkok</code>	[machine.c]
<code>spValueFromISRef</code>	<code>change</code>	[machine.c]
<code>spValueFromStackRef</code>	<code>addr</code>	[machine.c]
	<code>getvalue</code>	[machine.c]
	<code>lookup_address</code>	[eval.c]
	<code>eager</code>	[builtin.c]
	<code>touch</code>	[builtin.c]
<code>scheduleAdjacentTargets</code>	<code>schedule</code>	[eval.c]
<code>markRecursiveTargets</code>	<code>schedule_parents_of</code>	[eval.c]
<code>evalFormulaQueue</code>	<code>mark_changed</code>	[eval.c]
<code>invokeActionQueue</code>	<code>eval_formula_queue</code>	[eval.c]
<code>updateFormulaADR</code>	<code>invoke_action_queue</code>	[machine.c]
	<code>update</code>	[machine.c]
	<code>call</code>	[code.c]
	<code>execute</code>	[code.c]
<code>adjacentSourcesAreUpToDate</code>	<code>ready</code>	[refer.c]

Chapter 5

Problematic issues in dependency maintenance

This chapter pulls together three deep problems in dependency maintenance that run through the earlier chapters in this thesis, which previously had the status of ill-understood research problems. I make some specific proposals which help to deal conceptually with each problem. The proposals do not constitute detailed designs, but they do transform ill-defined abstract research problems into precise technical problems requiring an engineering solution that will be the subject of future tool development.

The three topics are those of concurrent definition maintenance, moding, and Higher-Order Definitions (HODs). The topics are somewhat intertwined. Each topic is treated below separately, but the discussion leads on from one to the next.

5.1 Concurrent definition maintenance

The design of a concurrent definition maintainer involves two principal issues. These issues are considered separately in the subsections that follow below.

1. Determining how to map evaluation agency to a set of ‘definition-agents’.
2. Specifying how the concurrent evaluation should be synchronised.

The job of a definition maintainer involves scheduling evaluations of definitions. Exactly when evaluations are scheduled depends partly on the choice of evaluation/storage strategy — evaluations can be performed on use, redefinition, or some mix of the two. (This was described in §2.2.1 and *am*, the DAM machine and EDEN are examples of each particular case.)

The specifics of a particular evaluation — the scheduling order of minor transitions within a major transition — depends on two things:

- A. the structure of the script digraph, as described by the corresponding level assignment¹, and
- B. which particular nodes are being redefined².

Within a major transition, there is sometimes potential for certain minor transitions (see §2.3.3) to occur concurrently. Specifically, within the set of nodes that require re-evaluation, all nodes that have the same level assignment may be evaluated concurrently. The potential for concurrent evaluation varies with the script. For example, N5A4ag (in Figure 3.28, see p.189) has a completely constrained evaluation ordering with no potential for concurrency. In contrast, after a change to the leaf in N5A4a (p.189), all the root nodes can be evaluated concurrently.

5.1.1 Mapping evaluation agency to definition-agents

Firstly, let us just consider issue 1. How many concurrent processes do we require and what part of the evaluation does each process perform?

Eden is a language that can describe both dependency and agency. One possible transformation of a definition to an “equivalent” evaluating agent was described using the Eden language in §4.3.3. When a set of definitions is considered, there are many possible ways that the transformation can be made. A good way to describe this is to extend the script digraph to take account of the evaluating agency of the definition maintainer. I describe this as adding *definition-agents* to the script digraph.

¹See Appendix §3.A, p.178 for Harary’s definition of ‘level assignment’ and §3.1.2 for Cartwright’s BRA — an algorithm based on Knuth’s topological sort algorithm that calculates an evaluation ordering consistent with that described by a level assignment.

²Section §4.3.2 gives a description of which nodes require re-evaluation after a change.

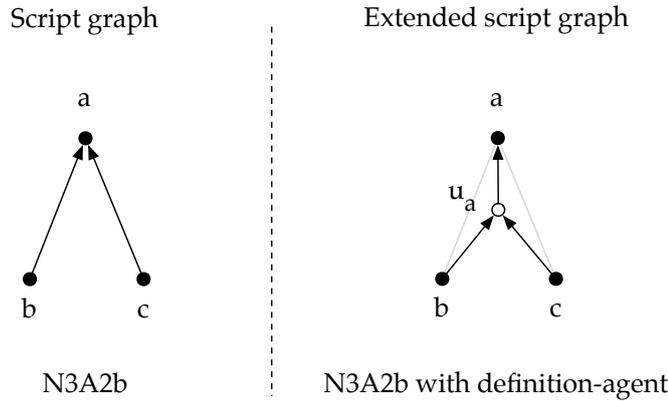


Figure 5.1: The script graph for `a is b+c`, together with an associated extended script graph devised by adding a definition-agent node

The script digraph for the one-line definitive script `a is b+c` is shown on the left of Figure 5.1 (reproduced from N3A2b in Figure 3.28 on p.187). Taking a ‘dependency-as-agency’ perspective (i.e. considering the way in which dependency is implemented through agency), we can transform the definition into Eden’s alternative³ triggered action form⁴:

```
proc ua: b, c { a = b+c; }
```

Now the agency involved in updating the variable ‘a’ can be represented by adding a new type of node, to be called a *definition-agent node*, to the script digraph diagram. The extended script graph diagram, shown on the right of Figure 5.1, reflects the dependency-as-agency perspective.

An extended script graph is a *bipartite*⁵ digraph. It has two types of node: the original *value nodes* and the new definition-agent nodes. It also has two types of arc. Arcs whose starting location is a value node (such as the arc `b – ua`) represent a *read* operation by the definition-agent. Arcs whose starting location is an definition-agent (such as the arc `ua – a`) represent a *write* operation by the definition-agent.

³But not completely equivalent — see §4.3.4.

⁴I use a subscript — which is not possible in the real Eden — to make clear which variable the action is updating.

⁵A graph G is bipartite if the nodes of the graph can be split into disjoint sets A and B such that each edge of G joins a node of A and node of B [Wil96, p.18].

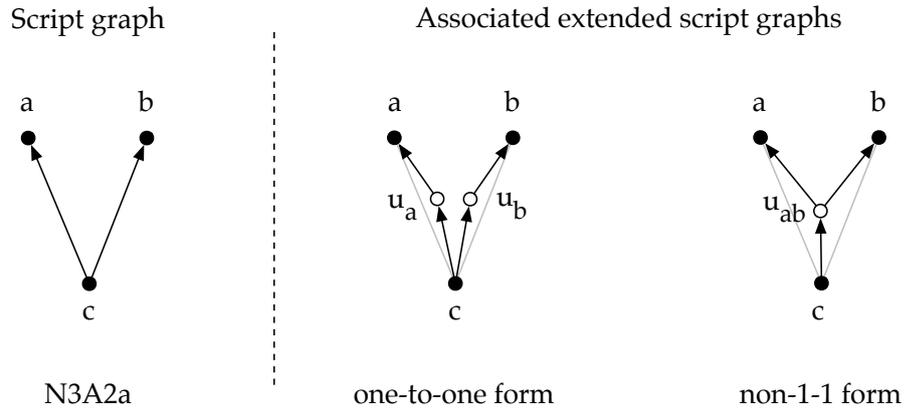


Figure 5.2: Two possible definition-agent arrangements for a simple two-line definitive script

In most non-trivial cases, there is more than one possibility for the total number and locations of definition-agents. Consider the script digraph shown on the left of Figure 5.2 (reproduced from N3A2a in Figure 3.28 on p.187), which represents the dependency structure in the two-line definitive script:

```
a is c;
b is c;
```

It is possible to maintain these two dependencies with either one or two definition-agents. Therefore, there are two possible triggered action forms of this definitive script: a *one-to-one* form where each definition is mapped to a definition-agent:

```
proc ua: c { a = c; }
proc ub: c { b = c; }
```

or a *non-1-1 form*, where more than one definition may be mapped to a single definition-agent:

```
proc uab: c { a = c;6 b = c; }
```

There are two corresponding extended script graph diagrams for these two scripts, shown on the right of Figure 5.2.

⁶These two operations need not be performed sequentially, but there is no way to specify this in the present Eden.

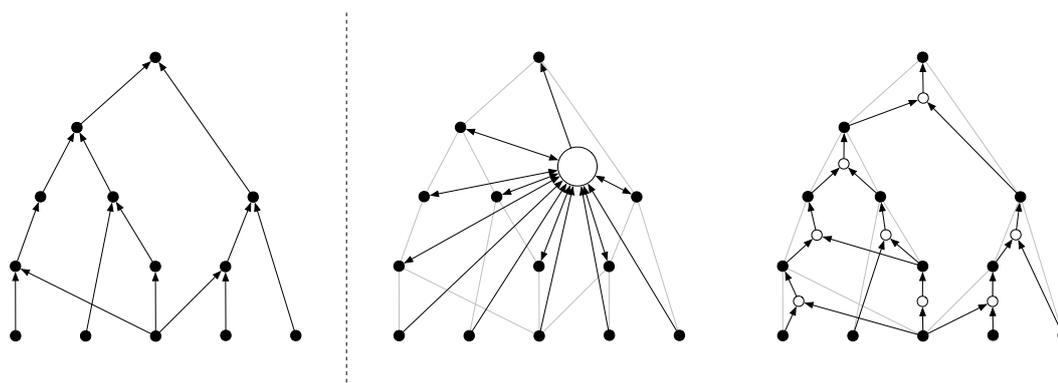


Figure 5.3: A script graph, the corresponding “monolithic” and one-to-one configurations of definition-agent nodes

Most definition maintainers can be considered to have only one definition-agent which is a “monolith”. In this type of analysis, the definition maintainer can be considered to be an agent of the form:

```
proc uall: all { schedule updates; do updates }
```

Figure 5.3 shows the distinction in diagrammatic form. A script graph is shown on the left. The middle of the figure shows the configuration of the single monolithic definition-agent in most definition maintainers. The monolithic definition-agent reads from leaves; writes to roots; and both reads from and writes to inner nodes. The DAM machine is an example of a DM with a monolithic definition-agent. In contrast, on the right, a one-to-one configuration of definition-agent nodes to non-leaf nodes is shown.

Each extended script graph describes a possible way of allocating update evaluations to agents. Figure 5.3 illustrates allocations at two extremes, where all updating is assigned to a single agent or where each update is handled by a separate agent. Many mappings between these two extremes are possible. The possible mappings of nodes of the script graph to definition agents provide us with a conceptual means with which to think about the number of concurrent processes required and what part of the evaluation each process should perform.

A previous reference that discusses the mapping of evaluation agency to concurrent processes is [Yun90, §7], where Y.W. Yung suggests (although not in these

terms) that a definition-agent matches the abstract view of a node in a multicomputer system, consisting of multiple nodes connected by a network, each comprising a CPU and some memory.

A definition in definitive languages can be viewed as a composition of *data* and *program*:

$$\text{definition} = \text{data} + \text{program}$$

where *program* is the method of evaluating the *data* and is expressed in mathematical terms. The *program*, theoretically, has no interference with other definitions since the only thing affected is the value (*data*) of the *definition*. This perspective on a definition matches the abstract view of the node in the multicomputer system:

$$\text{node} = \text{memory} + \text{CPU}$$

where *memory* stores *data* and *CPU* executes *program*. The data dependency of definitions describes the links of the nodes.

The dependency-as-agency discussion above has allowed us to go further than this “definition as node” perspective to show the wide variety of possible mappings of evaluation onto agents in concurrent dependency maintenance.

Y.W. Yung takes the discussion further in a different direction, pointing out that a single definition can be decomposed into sets of simpler definitions, which may create further potential for concurrent evaluation. In the terms used here, this corresponds to decomposing a single node in the script graph into a sub-graph of components. Figure 5.4 shows Y.W. Yung’s example of two different decompositions of the definition $f = ax^3 + bx^2 + cx + d$ (cf. Figure 4.16 on p.234, which illustrates different decompositions of output from the EDDI AOP).

On what basis are these various mapping decisions to be made? It seems that (similar to the basis for the decision of evaluation/storage strategy) the nature of evaluation and change in the application are important here. Although the script graph labelled B in Figure 5.4 has a larger total of level assignments and hence appears to have a smaller potential for concurrent evaluation, Y.W. Yung points out that in the special case where *a*, *b*, *c* and *x* are seldom changed but the value of *d* varies frequently, script B is likely to involve fewer evaluations than script A.

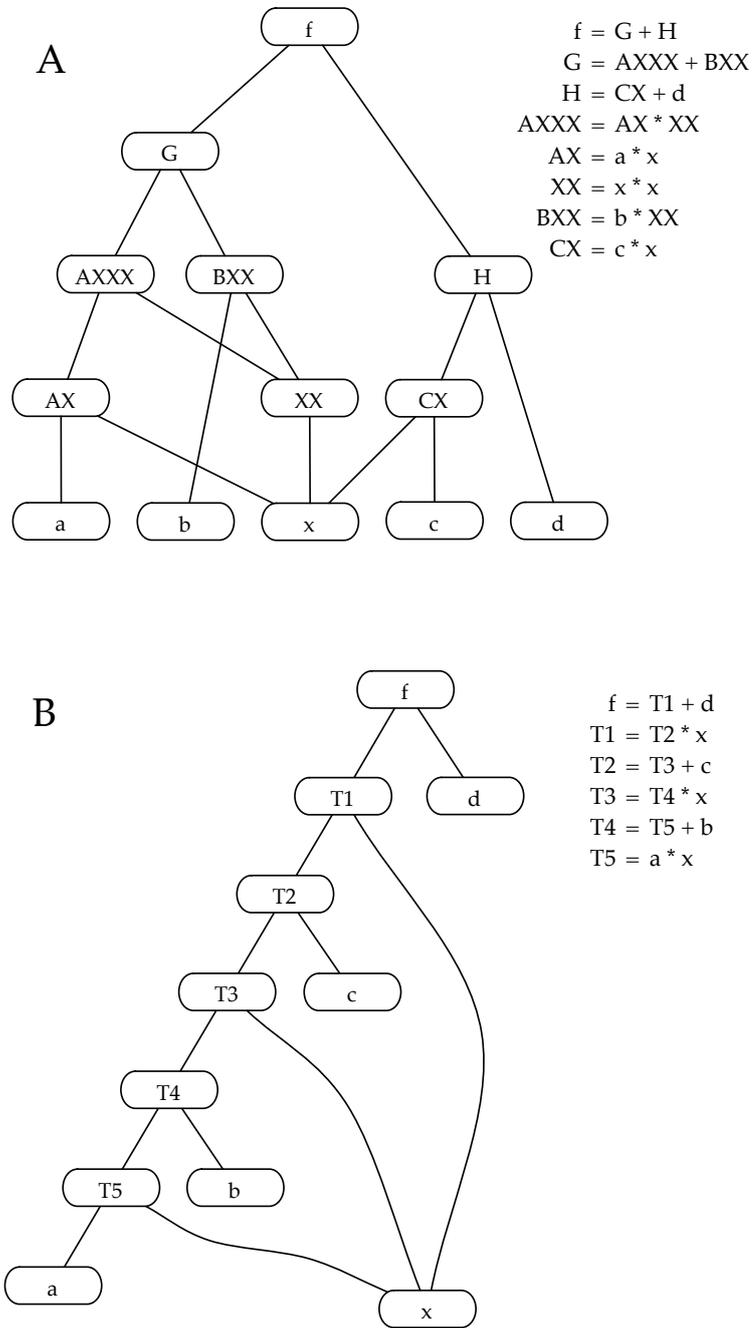


Figure 5.4: Decomposition of the polynomial definition $f = ax^3 + bx^2 + cx + d$ into two possible definitive scripts (from [Yun90, pp.97-98])

5.1.2 Synchronisation of concurrent definition-agents

The previous section has shown how evaluation can be segmented into concurrent parts. This section attempts to address the necessary synchronisation between those parts.

If we are motivated to use concurrency to speed up an existing sequential definition maintainer, then we do not need to examine the concept of dependency too closely — in this view, definitions describe potential parallelism in update. The necessary synchronisation is determined by the level assignments in the script graph. Within a major transition, necessary evaluations must be ordered by level assignment in order to produce values that are consistent with their definition. Within the set of nodes that require re-evaluation, all nodes that have the same level assignment may be evaluated concurrently.

A version of Cartwright’s JaM2 Java API implements a form of *concurrent update* along these lines. In JaM2, like the DAM machine, redefinitions are queued until an ‘update’ routine is invoked. The `update` routine locks access to the store of maintained state, checks the set of redefinitions for graph cycles, forms a schedule and then proceeds with evaluation, evaluating multiple definitions simultaneously in multiple threads where possible. When evaluation is complete, the store is then unlocked [Car04].

Y.W. Yung [Yun90, §5.3] also briefly describes a scheme for a concurrent definition maintainer employing message passing, nodes sending MARK, ACKNOWLEDGE, EVALUATE, QUERY and ANSWER messages to other concurrently operating nodes in order to propagate change.

Neither of these schemes considers the indivisible nature of dependency too closely, however. Beynon, Cartwright, Sun and Ward [BCSW99] contains the following characterisation of dependency, which is the starting point for consideration here of the necessary synchronisation (my emphasis):

A dependency is a relationship between observables that pertains *in the view of a particular agent*. . . changes. . . are *indivisible in the view of the agent*. That is: no action or observation on the part of the agent can take place in a context in which x has changed, but the dependants [targets] of x have yet to be changed.

The above statement represents a significant shift in emphasis in thinking about dependency within the EM literature. Usually (in documentation, teaching and papers), the *guarantees* that the definition concept gives are emphasised. For example, [Yun90, p.27], paraphrased:

No matter what the values of the source variables are, the value of a definition is *always* equal to its defining expression.

With this usual emphasis, $a \text{ is } b+c$ is interpreted as meaning that:

“ a is *always* $b+c$ ”.

The emphasis in DM implementation terms is then on automatic recalculation and propagation of change. A “concurrent update” implementation follows this emphasis: values are always consistent with definition, except when the `update` routine is in progress, when it is not meaningful to examine the state.

With the alternative, more recently topical emphasis, $a \text{ is } b+c$ is interpreted as meaning that:

“an agent that perceives the dependency $a \text{ is } b+c$ is restricted in some way whenever a is *not* $b+c$ ”.

The emphasis in DM implementation terms must then be on indivisibility in states and transitions, and synchronisation between agents. Dependency is created through definition-agent action perceived as indivisible by agents for whom the dependency pertains. Below, I describe my current understanding of this form of synchronisation of concurrent definition-agents.

First, let us define the roles that agents interacting in a definitive system can play. I separate the roles as much as possible into observation of state (O), change of state (C), and update of state (U). A ‘U-agent’ (shorthand for “an agent playing the U role”) is a definition-agent and is distinct (for the purposes here) from a ‘C-agent’. The U- and C-agent can be considered to be acting “inside” and “outside” the definition maintainer respectively. Various other separations of the roles are of course possible but these are the ones considered here.

Now we can define some protocols through which to achieve synchronisation. I define an observation to be bounded by ‘preO’ and ‘postO’ operations, and a change to be bounded by ‘preC’ and ‘postC’. Following the principles of indivisibility described above, it follows that change must exclude observation until the relevant definitive state has been updated. Therefore we can arrange for the C-agent to invoke the necessary U-agent in an ‘invokeU’ operation. The U-agent will signal completion of the update with a ‘postU’ operation. An observation “region” bounded by preO-postO operations must then not overlap with a change-update region bounded by matching preC-postU operations. This type of synchronisation is illustrated for the case of a one-definition script graph in Figure 5.5⁷.

The synchronisation described is an instance of the mutual exclusion problem [Dij68]. One way in which this can be solved is through the use of Dijkstra’s semaphore primitive. In this single definition script graph case, preO and preC can be implemented as $P(s)$ (potentially causing the agent to be blocked if necessary), and postO and postU as $V(s)$ (causing any waiting agents to be unblocked). Listing 5.1 shows a test implementation of this case, written in the language SR (Synchronizing Resources), “an imperative language for concurrent programming that provides explicit mechanisms for concurrency, communication and synchronisation” [AO93]. The code extends Figure 5.5 slightly by modelling two observing agents, O and O2. The agent O adheres to the interaction protocol, using the semaphore, and will never observe **a**, **b** and **c** in a state where **a** is *not* **b+c**. The agent O2 does not use the semaphore and hence it is possible for that agent to observe state inconsistent with the definition **a is b+c**.

Keeping to a one-definition script graph, but extending the example with more O- and C-agents would lead to a problem approximately⁸ equivalent to the classic readers/writers problem [CHP71].

⁷The conventions for the diagram are based loosely on [AO93, p.117].

⁸“Approximately” because it is unclear whether multiple concurrent writers should be allowed.

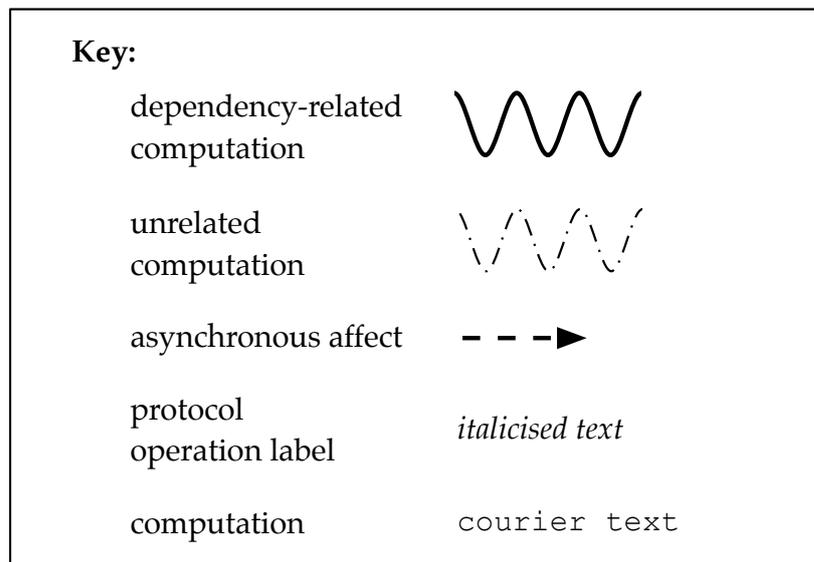
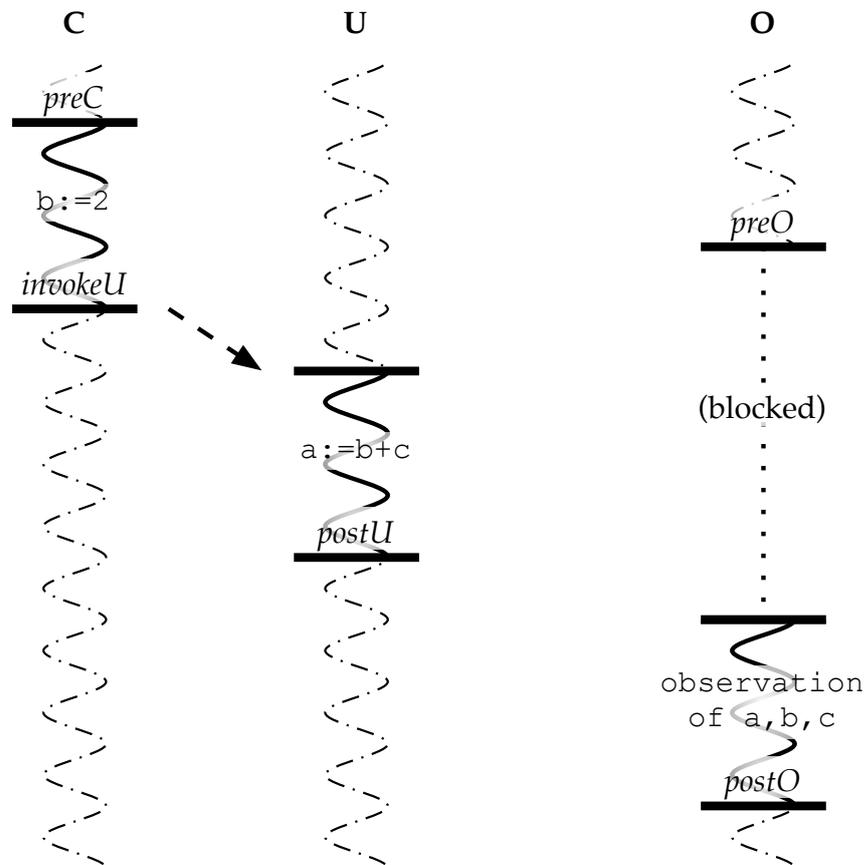


Figure 5.5: Synchronisation for indivisible observation of state in the single definition case

```

resource CU0()

  var a := 3, b := 1, c := 1    # initial values of dependency

  sem s := 1                    # maximum 1 agent acting simultaneously
  op U() {send}                 # U is to be asynchronously invoked

  # 0 observes a, b and c every so often and perceives a is b+c
  process 0
    fa i := 1 to 20 ->
      nap(int(random(100)))     # pause to introduce some non-determinism
      P(s)                       # pre0: block if C-U is acting
      write("0", a, b, c)      # observe a, b and c together
      V(s)                       # post0: allow C-U to act if blocked
    af
  end 0

  # 02 observes a, b and c every so often but does not perceive dependency
  process 02
    fa i := 1 to 20 ->
      nap(int(random(100)))     # pause to introduce some non-determinism
      write("02", a, b, c)     # observe a, b and c together
    af
  end 02

  # C changes b every so often
  process C
    fa i := 1 to 20 ->
      nap(int(random(100)))     # pause to introduce some non-determinism
      P(s)                       # preC: block if 0 is acting
      b := int(random(20))      # change b
      send U()                  # asynchronously invoke dependency update
    af
  end C

  # U is the agent that updates the dependency
  proc U()
    a := b + c                  # recalculate the dependency
    V(s)                       # postU: allow 0 or C to act if blocked
  end U

end CU0

```

Listing 5.1: SR code implementing the synchronisation shown in Figure 5.5

Extending the script graph beyond a single definition, to address a more realistic situation, changes the problem significantly. The additional complications relate to simultaneity of observation and change.

1. A definition describes a relationship between *multiple* nodes in the script graph. Simultaneity of observation of nodes and also, simultaneity of change to nodes thus become issues to be addressed. One way this can be added to the framework described so far is to extend the preO and preC operations, requiring information about the identity of the *set* of nodes that the agent is to observe or change.
2. The quote about dependency by Beynon *et al* cited above from [BCSW99] implies that dependency is a form of guarantee: when observed, values in a script graph are guaranteed to be consistent with their definition. The full implications of this statement go further: observation of the value of a node by induction entails observation of all values of the recursive sources of that node.
3. Immediately after a change is made to a value at a node in a script graph, the values of the recursive targets of that node potentially become inconsistent with their definition. Note that changes to the *definition* of a node have the same effect: although changes to a definition change the structure of the script graph, the values of the nodes *beneath* the change are unaffected (although these nodes may gain or lose a target reference).

The above discussion leads directly to a suitable strategy⁹ for synchronising observation, change and update of a script graph.

- Observation excludes change to recursive sources *below* the observed set of nodes, whilst observation is in progress.
- Change excludes observation of recursive targets *above* the changed set of nodes, until each value has been updated to be consistent with its definition.

⁹This is only one possible way of implementing the indivisibility requirement and may be unnecessarily restrictive.

Observation and change-update can therefore be thought of as placing restrictive ‘curtains’ over segments of the script graph. Observation of nodes (signalled by preO) causes a change-restricting curtain to be placed *below* the nodes to be observed. Once the observation within the curtain is complete, the curtain is atomically removed by postO. Change to nodes (signalled by preC) causes an observation-restricting curtain to be placed *above* the nodes to be changed. Once the change within the curtain is complete, the state can then be updated (concurrently where possible) by the U-agents, which signal completion of update of a node with postU. This then removes the restriction on observation. Therefore, as each node within the ‘curtain’ is updated, the curtain is lifted, progressively revealing the new state.

Appendix §5.A (p.328) shows an SR program that implements the multiple definition case. The program declares one semaphore for each definition in the script graph. The nodes beneath a particular set of nodes can be locked before observation of the set and unlocked afterwards. The nodes above a set of nodes can be locked before the set is changed. The values are then recursively updated, concurrently if a node has more than one target. The locks are removed as the state is updated. Although each node semaphore within a set must be locked sequentially, deadlocks do not occur as each concurrent (possibly competing) process makes the $P()$ and $V()$ operations in the same sequential order.

The “power” script example (Figure 3.2, p.115 *et seq.*) is implemented in the SR program with several concurrent processes that make various redefinitions and observe various node values. Some processes adhere to the interaction protocol and so always observe consistent state, some processes do not and so can observe inconsistent state.

Although the program functions as described, it is a prototype implementation only. Particular problems include that only one O-agent can be active on a set of nodes at any one time (a readers/writers solution would be better) and also the data describing the topology of the script graph (as opposed to the values) is not protected from concurrent update.

The above strategy is not the only possible way of meeting the indivisibility requirement. For example, it might be possible to allow change beneath an observed node, as long as the update that follows is prevented from propagating into the

nodes being observed. Another example of a possibly more general model would be as follows. The propagation of update following a change can be considered to form a ‘ray’ up the graph, originating from the change. Before a node in the path of the change is updated, it is in state S . After the ray has passed, it is in state S' . Observation of a set of nodes is permitted, so long as the entire set is either covered or uncovered by a ray (i.e. the entire set is in state S , or the entire set is in state S'). One complication here is that the covered/uncovered requirement applies only to nodes in the path of a change.

5.2 Moding

5.2.1 Problems with definitive lists

Section §3.5 discussed several problems related to the geometry of the DAM machine definitive store. There are significant problems involved in using data types whose representation is larger than one word, and the extension of the DAM machine to include a list data type poses very large problems. The `lookup` operator proposed by Cartwright ([Car99, p.159] — see §3.5.1) does not implement propagation of change from within the list and implementing this necessary dependency requires a fundamental rethink of the design.

Eden does include a list data type, and it is widely used. (For example, in the current `tkeden` implementation, the DoNaLD translator transforms each DoNaLD object to an Eden list, which is then used by utility routines written in Eden for rendering and other calculations.) However, there are various problems in the current EDEN implementation of lists. When used by procedural Eden code, treating a list as a conventional RWV, the implementation behaves as might be expected. When a list is referenced by a definition or defined as a FV, however, there are problems in every case.

Four problems are shown below to illustrate the issues as they appear in use of the Eden language. In each example, a comment starting with a `!` character indicates a problem.

Problem 1: Observation of a list FV causes evaluation

```
1  %eden
2  l is [1, g(x)]; /* g is undefined */
3  writeln(l[2]); /* error: func "g" needed */
4  writeln(l[1]); /* ! error: func "g" needed */
```

In this example, it is not possible to observe the value of the second element of the list (at line 3) as the function `g` has not yet been defined. Observation of the first element (at line 4) should however be possible, but the current EDEN implementation gives an error. Generally: observation of an individual list element causes the *entire* list to be evaluated.

Problem 2: Difficulty of redefining portions of a list FV

```
1  %eden
2  a = 1; b = 2;
3  l is [a, b];
4  l[1] is 3; /* ! parse error */
5
6  ?l;      /* gives "l is [a,b]" */
7  l[1] = 4; /* this input is accepted, but now the entirety
8            of l is a RWV, not a FV */
9  ?l;      /* ! gives "l = [4,2]": the dependency between
10           a, b and l has been lost */
11
12 proc p: l { writeln("l has been changed"); }
13 b = 5;    /* ! triggered action p is not invoked */
```

Firstly, line 4 of this example shows that redefinition of part of a list FV is not implemented. Secondly, lines 6–13 show that making a procedural assignment to one element of a list FV makes the entire list a RWV, which is often not what is desired. Generally: it is not possible to redefine portions of a list FV¹⁰.

¹⁰It is possible through use of the Eden language to process the existing definition as a string, make the necessary changes and re-parse the result using the `execute()` command, but this is inelegant and a fully robust solution is complex.

Problem 3: Indiscriminate change propagation with list FVs

```
1  %eden
2  l = [1, 2];
3
4  head is l[1];
5  proc p: head { writeln("head has been re-evaluated"); }
6
7  l[1] = 3; /* correct invocation of triggered action p */
8  l[2] = 4; /* ! incorrect invocation of triggered
9             action p */
```

Here, the FV `head` is observing the first element of the list. When the first element of the list is changed at line 7, the execution of triggered action `p` reveals that `head` has been correctly re-evaluated. However, when the second element of the list is changed (line 8), the triggered action is again executed. Generally: change to an element of a list causes re-evaluation of dependencies observing *any part* of the list.

Problem 4: “Phantom” graph cycles in list FVs

```
1  %eden
2  l is [6, l[1]]; /* ! error: cyclic dependency detected */
3
4  l is [6, a];
5  a is l[1];      /* ! error: cyclic dependency detected */
```

In this final example, two attempts are made to define the second element of the list to be the same as the first element, which is defined to a literal value. There are no cycles in the script graph here if `l[1]` is interpreted as referring to the first element of the list *as an individual*. However, the current EDEN implementation makes no provision for such an interpretation and so detects a graph cycle. Generally: “phantom” graph cycles can be inappropriately detected when references to the individual elements of list FVs are made.

At the root of the four problems is the fact that the current EDEN implements a reference to a list element using a function. An element reference is effectively translated internally¹¹ to a functional representation:

$$v \text{ is } l[i] \rightarrow v \text{ is } f(l, i)$$

Statically, this translation is “correct”: reference into a list is a mathematical function in the sense that the function f maps the l and i arguments in a uniform way to the result value v :

$$f : (l, i) \mapsto v$$

The mapping is consistently applied whenever invoked — in other words, definitions are referentially transparent. However, the functional interpretation of the element reference does not capture one essential ingredient of this kind of reference.

If the functional interpretation is taken, then changes to the arguments of the function imply a possible change to the value of the function. If the offset argument i is changed, then certainly the value v will reference a different element of l and so may change. But if an element of the list argument l that is *not* referenced by the current offset argument i is changed, then the value v will certainly *not* change. The functional interpretation applied in this way is simply not specific enough.

The functional interpretation of reference to an element is at the root of the four problems in the current EDEN implementation described earlier. The general lesson here may be that when considering dependency over composite types, the functional abstraction applied at the composite level is inappropriate, as it does not capture an essential aspect of the nature of reference.

Considering propagation of change rather than functional abstraction has revealed the above problems. Further consideration of the issues of lists in this way leads to the following idea. When a change is made, the old value can, in some situations be re-used in order to assist with calculation of the new value. This would

¹¹Although note that this is not a translation involving strings and notation — this occurs at the level of EDEN’s virtual machine.

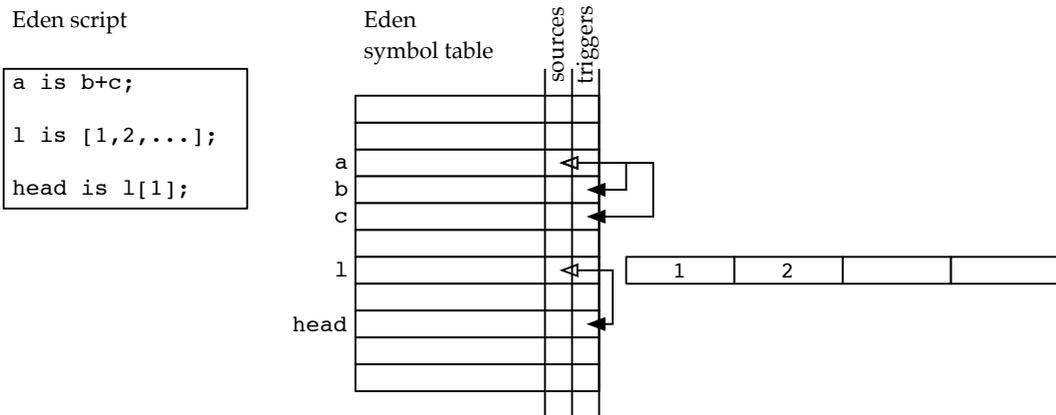


Figure 5.6: EDEN implements lists “horizontally” in the symbol table

improve update efficiency and gives no semantic problems. For example, the sum of a list can be re-used if one element changes:

$$\text{composite_sum_new} = \text{composite_sum_old} + (\text{changed_element_new_value} - \text{changed_element_old_value})$$

As it is the references to composite data structures in EDEN that lead to problems of specificity of reference, one way to work around the problem is to eliminate such data structures. I have constructed a solution in EDEN which involves reducing all composite data structures to multiple atomic variables. In terms of the EDEN symbol table, the problems outlined in this section occur because lists are implemented “horizontally” — all the elements of the list are located conceptually at the location of the list identifier, as illustrated in Figure 5.6.

A macro translator, named the Eden “Symbol Lists” translator was written for EDEN in Eden, using the regular expression facilities and generalised notation framework (described in §4.2.4). The translator performs macro transformations on Eden input before it is parsed, changing language constructions involving lists into a form where the individual elements of the list are each stored in their own symbol. It is activated by changing the notation context to `%edens1`. The component parts of a list ‘1’ with three elements will be stored in the symbols 11, 12 and 13. The length of the list is stored in another symbol, 11. The exact transformations used are illustrated in Figure 5.7.

```

l = [a, b, 42]; → edensl_assignconstruct("l", [a, b, 42...]);
l is [a, b, 42]; → edensl_defineconstruct("l", ["a", "b", "42"...]);
l[4] = a; → edensl_assignelement("l", 4, a);
l[4] is a; → edensl_defineelement("l", 4, "a");
l# → ll
l[i] (on RHS) → li
append l, v; → edensl_append("l", v);
delete l, i; → edensl_delete("l", i);
insert l, i, v; → edensl_insert("l", i, v);
shift l; → edensl_shift("l");
l = ... → edensl_assign("l", ...);
l is ... → left unchanged

```

Figure 5.7: Transformations implemented by the `%edensl` translator

As the macro transformation causes the list components to be stored separately, after transformation, there is no way of observing the list as a whole. This problem is solved by the introduction of a definition with a name as in the original construction, defined to be a list FV, naming each individual element as a component. This definition needs reconstruction when the list changes in length, and this is done by a triggered action. Figure 5.8 shows an example transformation and a representation of the resulting symbol table, where it can be seen that the list is now stored “vertically”, each element in an individual symbol.

The `%edensl` macro translator solves all four problems mentioned earlier. It can even be configured to “replace” the standard `%eden` notation, in which role it is transparent to the user. (The original Eden is still available by using the notation context `%eden0`, as illustrated in the script in Figure 5.8.) The decrease in performance that the macro translator causes may actually be quite small, since once the transformation and parsing is complete, EDEN stores the virtual machine opcodes — re-parsing is not necessary during machine execution. However, a major limitation of the macro translator is that the blocking of the regular expression transformations is on a per-line basis, and the EDEN `execute()` routine used to pass transformed output to the Eden interpreter does not accept partial input. The translator therefore currently fails to process multi-line procedures correctly.

Script

```

%edens1
l is [a,b,3];
head is l[1];

->
  %eden0
  l1 is a;
  l2 is b;
  l3 is 3;
  l1 = 3;
  l is [l1, l2, l3];
  proc l_constructwhole : l1 {
    /* constructs new definition of l
       when l1 changes... */
  }

  head is l1;

```

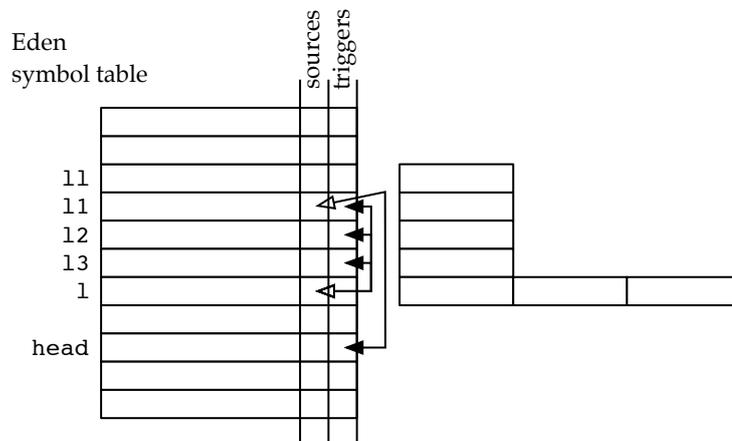


Figure 5.8: The `%edens1` translator transforms references to lists, causing EDEN to locate them “vertically”

5.2.2 Mezziani’s DENOTA and the “mode of definition” of a variable

Contemporary with the initial development of EDEN, DENOTA (for DEfinitive NOTations) was developed by Samia Mezziani, who described it in her MSc thesis [Mez87]. The aim was to implement a tool to handle data abstraction within the definitive programming paradigm. Of interest in this section is the concept of *mode of a variable*. The concept had some precedent in Beynon’s ARCA notation, first described in [Bey83] (which is the first formal publication about definitive concepts, even predating that term). However, [Mez87, §4] is the only publication with a significant treatment of the “moding” topic¹².

“Moding” can be motivated as follows. In interaction using a definitive notation that supports lists, a variable can be defined to be a function that evaluates to a list, or it can be defined using a list constructor.

After a variable has been defined functionally, later in the interaction, the variable can be redefined. However, later in the interaction, the *components* of the variable cannot be redefined. As is illustrated by the Eden example associated with Problem 2 in §5.2.1, partial redefinition is not possible in general, since (as we show below) such a redefinition would correspond to a “reprogramming” of part of the function in some way.

After a variable has been defined using a list constructor, later in the interaction, similarly, the variable can be redefined. This time, the components of the variable *can* be redefined: partial redefinition is possible. Such a redefinition does not constitute a “reprogramming” of the constructor — the constructor has been used to create the initial “shape” of the list and now the shape, not the constructor, remains.

The following examples are intended to illustrate this point. In order to give meaning to the examples, I have used the Eden language¹³. However, the examples use the Eden syntax as a language independently of our current implementation — as described in the previous section, the current EDEN implementation does not deal well with definitive lists. Also note in particular that this discussion relates

¹²[Geh95, Bir91, Car99] variously describe moding but do not add any information beyond that given in [Mez87].

¹³The examples are my own — [Mez87] does not illustrate this basic point with an example and the notation used there appears to be based partially on LISP.

only to definitions: the examples use Eden FVs exclusively — RWVs, created by assignment, do not appear.

First, we illustrate a functional definition and the impracticality of then interactively redefining it component-wise.

```
func f { return [1,2,3]; }
l is f();
l[1] is 4; /* ! not possible */
```

The redefinition of the first component of the variable `l` could imply a redefinition of the function `f`, perhaps to the following.

```
func f { return [4,2,3]; }
```

In the more general case (for example if control flow is used within function `f`) then this modification is not possible to determine automatically.

Alternatively, the redefinition could imply the use of an additional function layered on top of the existing function `f`, modifying the first component, as follows.

```
func g { q = f(); q[1] = 4; return q; }
l is g();
```

However, successive uses of such redefinitions to one variable interpreted in this way would lead to many `g`-style functions building up in the state, effectively representing the entire history of the interaction with that variable.

Use of a list constructor implies no such problems:

```
m is [a, 2, c];
m[1] is 4; /* OK */
```

Given that the variable `m` has been initially defined using a constructor, the redefinition of the first component of `m` implies the following redefinition:

```
m is [4, 2, c];
```

which takes the history of the interaction into account but the result of which is still meaningful statically.

The above discussion exposes serious problems in interpreting the way in which definitions match variables to formulae. Mezziani's proposed solution to this problem

is to introduce an auxiliary definitive notation in which the mode of definition of variables can be declared. In effect, definitive principles are being used to supply the meta information needed to disambiguate reference and constrain redefinition. A significant aspect of using an auxiliary definitive script to define mode is that the relationship between variables and their definitions can be as flexible and dynamic as value definition in a definitive script.

The application of auxiliary definitive notations of this nature is not confined to handling the mode of definition of variables. For instance, Meziani also proposed that similar principles could be adapted to provide information hiding in definitive scripts [Mez87, p.75]:

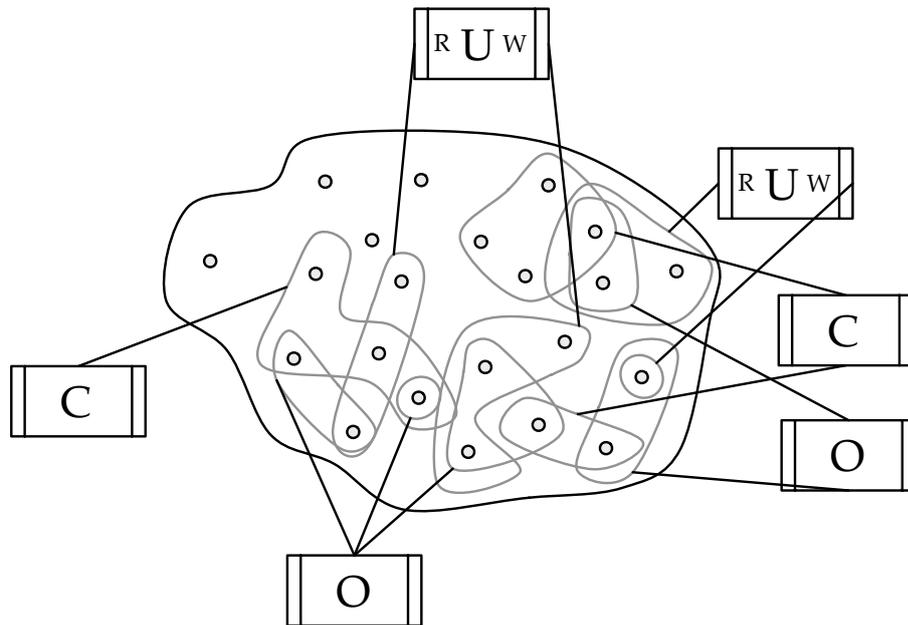
Definitive notations could be augmented to support information hiding. For this, two facilities could be used, namely a reference moding, and a definition moding of variables. The latter would describe the way in which variables are referenced and defined. The former could describe the way definitive variables could be viewed. For instance, a variable of reference mode `abst list` could have hidden components. These concepts would also be useful to express semantic rules such as: the `tail` could apply on `abst list` variables. These potential facilities would provide stronger typing of expression analogous to Abstract data types, and Object-oriented programming paradigms.

Here, moding has a more generic meaning than has been applied so far, since it can be used separately for “reference moding” and “definition moding”.

Our current techniques for implementing definitive notations are not sophisticated enough to support such features.

5.3 The tri-box framework for higher-order definitive state

This section sketches a “tri-box framework” for higher-order definitive state that I have developed in response to some of the issues raised in this thesis. The first subsection briefly summarises the motivations for the framework, the second subsection sketches the framework concept and gives some examples and the final subsection outlines issues for implementation raised by the framework.

Figure 5.9: O-, C- and U-agents interacting with the state S

5.3.1 Motivation

Section §5.1.2 proposed three roles for agents interacting in a definitive system, in order to consider the types of synchronisation that a concurrent definition maintainer might use. Each of the O-, C-, and U-agent roles (for observation, change and update respectively) observes or acts on a subset of the state within the system. Here, we name the state S (following the discussion of transitions from an initial state S through intermediate S^* states to a resultant S' state in §2.3.3). In an LSD account, each agent *perceives* values — there may arguably be no objective state S . In this section, we assume that the authentic values of observables can be directly acted upon or observed by O-, C- and U-agents: the framework is similar to the ADM in this respect. Figure 5.9 illustrates several agents playing different roles interacting with the state S .

In the EM group, we conventionally describe the state S by the use of a definitive script written in a definitive notation. However, much of the work described in this thesis hints that a symbolic definitive script is in many ways an inadequate framework in which to discuss the propagation of change in its full richness. For example,

previous sections have made the following points about symbols in a definitive state.

- A definition-agent writes to only one symbol (§4.3.4, §4.3.5). This constraint on definition-agent action gives traceable meaning to the state.
- But a U-agent in the extended script digraph can write to more than one symbol (§5.1.1). This can be necessary for implementation efficiency reasons with current machine architectures, where there is not enough parallelism available to make a one-to-one mapping of U-agents to the script graph. It may also be necessary for semantic reasons in the absence of notations describing powerful higher-order dependency, which could for example be used to describe the dependency involved in updating the screen (§3.5.4).
- The mapping of a symbol to locations of words in store is a complex distraction at the level considered here. One symbol can correspond to more than one word in store, depending upon type (§3.5.1). The symbol table is another level of indirection (§3.4.3), itself a higher-order dependency. We wish to use dependency in the symbol table to create dependency-driven parsers (§4.2.5).
- The functional abstraction of reference implied by symbols in a definitive script causes many problems in the use of definitive lists (§5.2.1).
- The dependency described by a definition need not be objectively perceived — different agents may have different perceptions (§5.1.2).

The presence of symbols implies that there is one objective understanding and state of each symbol. However, in our natural language we use symbols for *reference* only and in many instances we start from an assumption of subjective understanding.

Are there atoms in definitive state at all? We may wish to introduce variously-sized atoms in order to cope with different data types (§3.5.1); atoms that have hidden composite structure (one particular mode of definition — §5.2.2); structures formed from atoms (another mode of definition), perhaps with some values hidden from certain observers (as Meziani suggests in her “reference moding” concept). If there are atoms, are they ordered in any way? We may wish for various forms of ordering: words in store can be considered to be ordered by a single dimension;

the screen naturally has a two dimensional ordering (§3.5.4); spreadsheets have an ordering that is at least two-dimensional (and possibly three-dimensional, if for example Excel’s ‘sheets’ are taken into account). Pursuing the connections developed in [Bey97, Bey99, Bey03], the nature of definitive state may be likened to that of William James’s “unfinished pluralistic universe”¹⁴. Notice that Figure 5.9 illustrated S as an unordered set of atoms of which there was a single basic type.

In order to tackle some of the above issues, this section moves away from the notion of the symbolic definitive script. Instead, below, we take S to be a sequence of atomic ‘boxes’ in store.

The framework described below differs from the DAM machine as described in Chapter 3. Although the DAM machine may appear to have been designed with a primary focus upon dependency between atomic words of store, it is actually an ‘implementation’ of the DMM. The DMM is a formalisation of the Low Level Definitive Notation (LLDN) concept, a symbolic notation describing dependency between a set of atomic integer values, and hence the symbolic influence on the DAM machine design is strong. LLDN can describe dependency only between single words in store — neither data types larger than a word (§3.5.1) nor lists (§3.5.2) are possible. As the basis of LLDN is a *set* of integer values, identities are not structured in the DAM machine, as they would be in an authentic “generalised spreadsheet” (a term previously used to describe definitive scripts — see §3.5).

The focus on the sequence of atomic ‘boxes’ in store below also means that we must treat the concept of the script graph with caution. A script digraph (see Appendix §3.A, p.178) describes the propagation of change required in the symbolic script structure. An extended script graph describes how the propagation of change is mapped to updating agents. But if we wish to consider higher-order dependency, the script graph is itself subject to dependent change¹⁵. The change may be limited to just the arcs in the graph (for example, in the case of the `if HOD`, described in §4.3.7) or it may also involve the addition or removal of nodes (for example, in the

¹⁴Wild describes this universe as having “. . . aspects of unity, relations which hold different members of this collection together. But there are also aspects of diversity and independence. As we live through this empirical world, it is ‘like one of those dried human heads with which the Dyaks of Borneo deck their lodges. The skull forms a solid nucleus; but the innumerable feathers, leaves, strings, beads, and loose appendices of every description float and dangle from it, and, save that they terminate in it, seem to have nothing to do with one another’ [Jam12]” [Wil69, p.391].

¹⁵This thesis is seemingly the first writing to describe Higher-Order Dependency in this way.

case of ADM entities instantiated or deleted by the redefinition of a LIVE variable, mentioned in §2.1.1).

By moving away from the notion of the symbolic definitive script and taking S to be sequence of atomic ‘boxes’ in store, we can concentrate in this section on the synchronisation of mechanism and perception involved in interaction with meaningful state. The mechanism here is the action of U-agents in response to change initiated by C-agents. The perception is achieved by synchronisation of O-agents with respect to C-U-agent action. O-, C- and U-agents all act on or observe a subset of S . The desired synchronisation and interaction with subsets of S is illustrated in Figure 5.10.

The tri-box framework does not attempt to solve all these issues. However it does seem to clarify:

- the concept of U-agents that write to more than one atom (for example, the screen);
- the problem of indiscriminate change propagation associated with definitive lists (Problem 3 described in §5.2.1);
- restricted forms of higher-order dependency (involving dynamic script graph arcs), and
- the synchronisation required for subjective dependency.

The tri-box framework focusses on *propagation of change* within the current state, rather than *evaluation* of a script. The framework has emerged by asking the questions: *What is the minimum information we need to implement a concurrent definition maintainer?* and *What is the simplest way to organise the information?* In asking such questions, we have moved away from the symbolic emphasis of scripts, but hope to return with some insights for our symbolic notations.

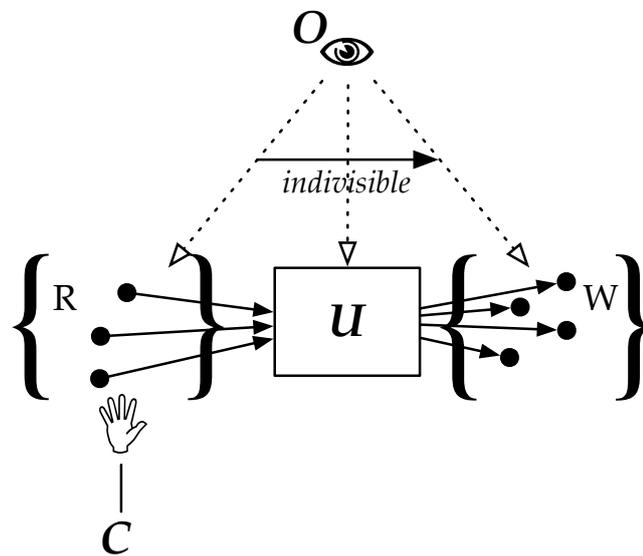
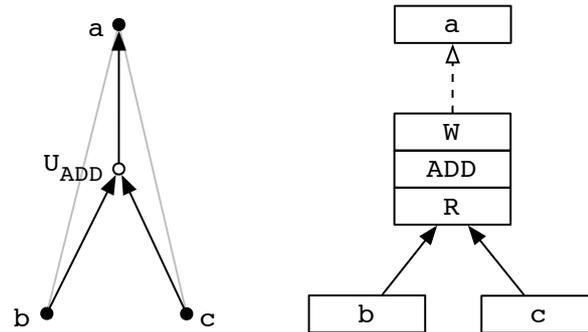


Figure 5.10: O-, C- and U-agents observe & act on subsets of S and are synchronised

Figure 5.11: Script graph and tri-box diagram for $a \text{ is } b+c$

5.3.2 Concept and examples

The tri-box framework is a “visual formalism” [Har88] describing the propagation of change within a sequence of atomic ‘boxes’ in store. It can be viewed as a generalisation of the extended script graph (described in §5.1.1).

The extended script digraph for the definition $a \text{ is } b+c$ and the corresponding tri-box diagram are shown in Figure 5.11. The value nodes of the extended script digraph have been replaced by ‘value boxes’, each holding a value in the store S . A value box may or may not correspond to a word in store and may be of fixed or variable length — these are matters of implementation.

The U-agent node in the extended script digraph has been replaced by three vertically joined boxes, termed a ‘tri-box’. From top to bottom, the three boxes in a tri-box are named the ‘W-box’, the ‘U-box’ and the ‘R-box’. The values contained in the boxes describe the ‘W-set’, the identity of the update operator and the ‘R-set’ respectively. The W-set is the particular subset of boxes in the store S that the U-agent may write values to, and similarly the R-set is the subset of boxes that it may read values from. The subset of S referenced by each U-agent can therefore be specific to each U-agent. The framework captures the notion of subjective reference as opposed to the identification of objective symbols.

The value of the W-set is held conceptually in the (top-most) W-box, and is diagrammatically represented by drawing ‘W-coloured’ arcs from the W-box to the boxes included in the W-set. Similarly, the value of the R-set is held conceptually in the (lowest) R-box. The value of the R-set is represented diagrammatically by

drawing ‘R-coloured’ arcs *from* the boxes included in the R-set to the R-set box. The three adjacent boxes and arcs are drawn this way so as to mirror the conventional geometric layout of the corresponding script graph, where sources of the U-agent are conventionally placed below, targets above, and arcs describe propagation of change.

The dependency in the system is thus described by the information held collectively in all the tri-boxes. Following Slade [Sla90], this information is known as D (see §2.1.2). The information is used to coordinate and synchronise the O-, C- and U-agents, as outlined in §5.1.2 and illustrated in Figure 5.10.

If the boxes making up the tri-boxes are located in S along with the ‘value boxes’ (this organisation can be succinctly termed ‘ D -in- S ’), the values contained in the tri-boxes may be written to by U-agents. In this case, the script graph itself is maintainable by dependency. The framework therefore provides a conceptual means with which to describe higher-order dependency.

There is one rule restricting the topology of a tri-box diagram, which follows from the discussion in §4.3.4 of constraints upon definition-agent action:

Each box in S must be referenced by *at most one* W-set.

Alternatively stated, considering the W-sets as sets of arcs (as drawn in Figure 5.11), each box in S must have at most one incoming ‘W-arc’. This rule ensures that no word in S is subject to change from two or more independent agents. It is then possible to trace effect back to cause.

Some examples to illustrate how the tri-box conceptual framework can be applied are given on the following pages. The next subsection then discusses implementation of the framework.

Example 1: The “power” script — illustrating many possible U-agent configurations

Two non-trivial examples of the tri-box diagrammatic form are shown in Figure 5.12, which depicts two configurations for maintaining dependency in the “power” script (a running example in this thesis, which first appears in §3.2.4). The figure shows the “power” script, the script graph and two possible tri-box implementations of the script graph. The tri-box implementation shown on the left has U-agents in one-to-one correspondence with nodes (the “one-to-one configuration” — see §5.1.1). Notice that there are two distinct **ADD** update operators, for reasons discussed further in the next subsection. The tri-box implementation shown on the right uses a single U-agent. This is a ‘monolithic’ configuration (see §5.1.1).

The tri-box framework can represent the many possible mappings of script graph nodes to updating agents which are possible between these two configuration extremes (*cf.* the discussion of extended script graphs in §5.1.1). Each possible mapping has differing potential for concurrent update.

Script

```

a = 3
b = 10
c = 7
d = 9
e = add(a,b)
f = add(b,c)
g = times(e,f)
h = max3(g,f,d)
i = power(h,a)
    
```

Script graph

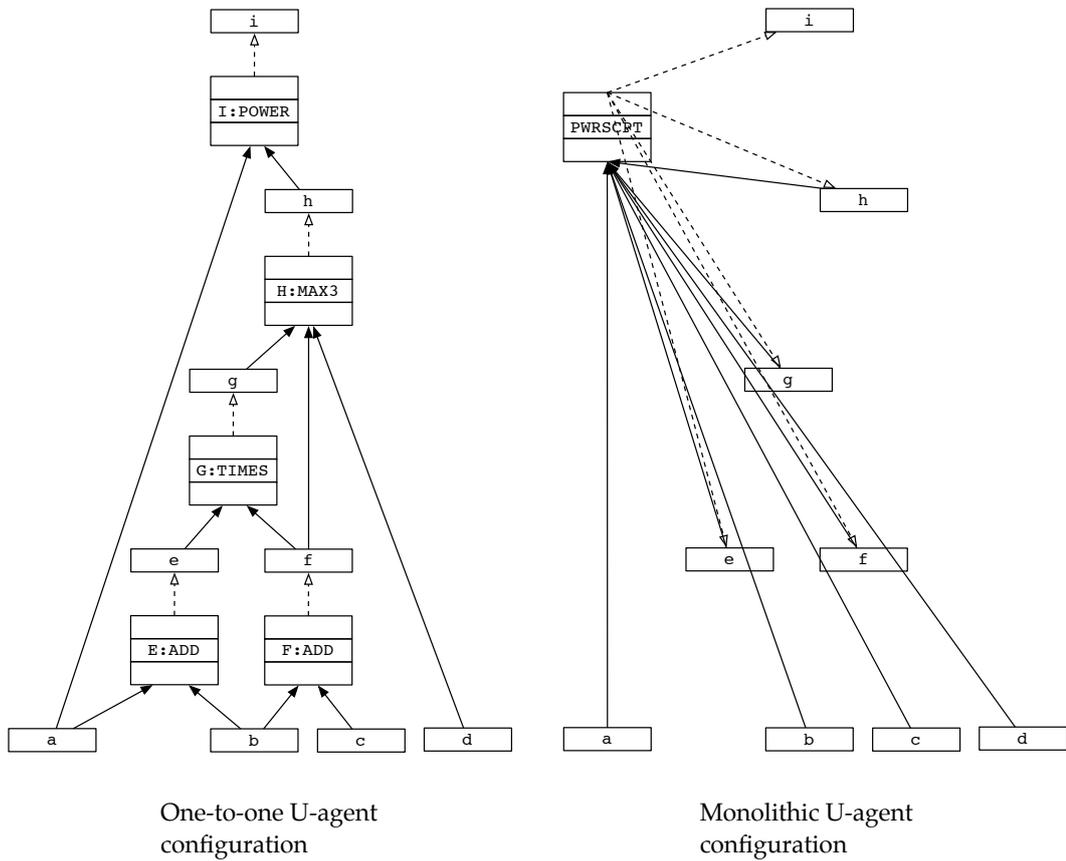
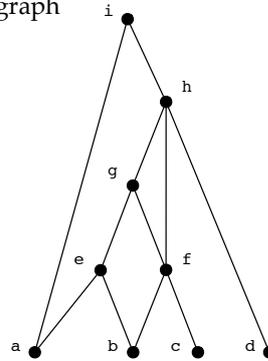


Figure 5.12: Tri-box diagrams of the “power” script

Example 2: Character glyphs — multiple W-set items and hardware

This example demonstrates some of the potential for using W-sets containing more than one reference, and the ability of the framework to represent dependency that is present externally to a definition maintainer. The example is related to the one given in §3.5.4, where a character pattern dependent on a character code word was made to appear on the screen.

At the bottom left corner of Figure 5.13 a single U-agent with the update operator `CHR_G` is shown. This U-agent reads from a single value box containing a character code (labelled `CODE`). The `CHR_G` (“character code to glyph”) operator then internally calculates the appropriate character glyph and writes to eight value boxes (labelled `PW1` to `PW8`).

The configuration of the DAM machine described in §3.5.4 exploited the video hardware of the machine to directly render definitive state. In this configuration, the video hardware can be considered to be a U-agent, as on-screen state is indivisibly (at the level of human perception) related to video RAM state. This U-agent can be represented as a tri-box as shown in the figure, where the “update operator” is labelled ‘(HW)’. As shown in the figure, the single box `PW2` (in implementation, a 32-bit machine word) corresponds to 32 on-screen pixels¹⁶. Only one box to pixel region mapping is shown for clarity. Multiple mappings could be achieved by adding more tri-boxes or by extending the R- and W-sets of the existing ‘(HW)’ tri-box.

¹⁶In the black and white graphics mode used by the DAM machine in single-tasking mode, one 32-bit machine word corresponds to 32 on-screen pixels, each bit representing one binary pixel state.

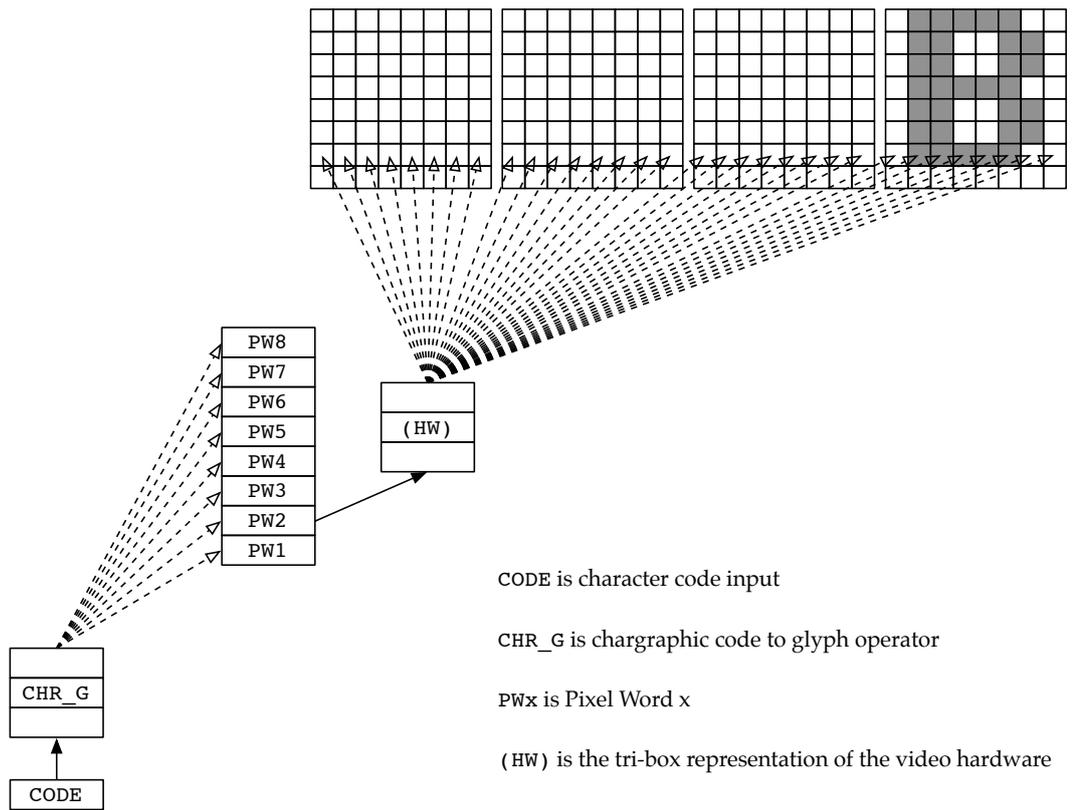


Figure 5.13: Tri-box diagram showing an on-screen character glyph representation linked by dependency to a box containing a character code

Example 3: Observation of overlapping subsets — discriminate references

This example relates to the problem of indiscriminate change propagation in EDEN, discussed previously as Problem 3 in §5.2.1. In the present EDEN, change to any element of a list causes re-evaluation of dependencies observing *any part* of the list, whether the changed element is observed or not. The problem is due to the reliance in Eden on a single symbol to represent the entire list, and the functional abstraction of reference employed, such that the list element $l[1]$ is represented internally as the functional $f(l, 1)$.

In the tri-box framework, references (to subsets of S) are specific to each U-agent — they are not objectively encapsulated in a symbol. Figure 5.14 shows two U-agents observing overlapping subsets of S . Due to the specificity of reference of the R-set, there is no need for the implementation to invoke the updating agent U2 when a value box not contained in its R-set (such as the box marked T) is changed.

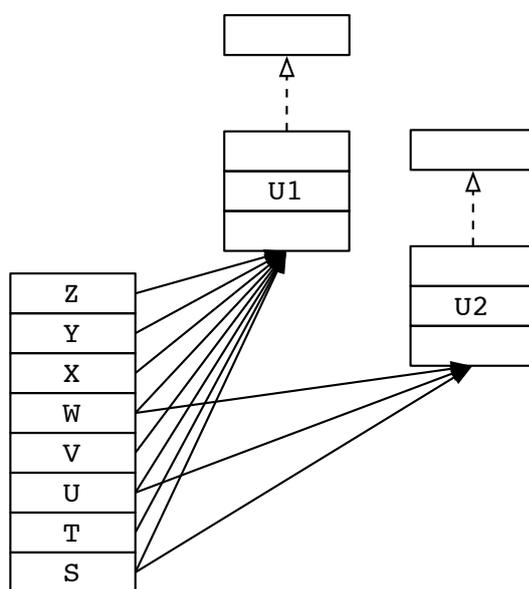


Figure 5.14: Two U-agents observing sub-sets of a list

Example 4: The if operator — Higher-Order Dependency

Section §4.3.7 briefly described the Eden ternary `if` operator and problems with its use in a FV due to the functional abstraction of reference. Effectively, the Eden definition:

```
v is b ? x : y;
```

is represented internally as the functional:

```
v is f(b, x, y);
```

but a representation correctly reflecting the propagation of change required would, depending upon the value of `b`, list *either* `x` or `y` as a source of `v`, but not both. The presence of both `x` and `y` as sources in the current implementation can cause “phantom” graph cycles to be detected (see Problem 4 in §5.2.1).

The ternary `if` creates a simple higher-order definition when used in a formula¹⁷. In this example, the value of `b` affects the arcs required in the script graph. Considered this way, the script graph arcs can be reconfigured when necessary using the following Eden triggered action, presented in §4.3.7:

```
proc cv: b { if (b) { v is x; } else { v is y; } }
```

The same solution can be modelled in the tri-box framework if we locate *D-in-S*. The W-set of the `WRITEREF` tri-box in Figure 5.15 references the R-box of the `COPY` tri-box. The `COPY` update operator implements the simple identity function: the `COPY` tri-box simply copies the value referenced by its present R-set to the value box `v`. The R-set of the `COPY` tri-box is made to reference either `x` or `y` by the `WRITEREF` tri-box (the alternative possibility to the current state is denoted by the grey line drawn from the `y` value box). The `WRITEREF` update operator has as output the value of the R-box representation denoting either a reference to the value box containing `x` or that containing `y`, the choice of output depending upon the boolean value of the input `b`.

¹⁷Although not when used in an assignment, since this does not create a definition.

The tri-box solution to this problem shown in Figure 5.15 can in principle be examined statically to trace cause and effect. There is an answer to the question “Why is the value of v currently that of x ?” An answer to this question in the Eden triggered action solution depends upon the implementation making a record of the identity of the last action to change the definition of v .

The `if` HOD example involves reconfiguration of only arcs in the script graph (R-sets in a one-to-one tri-box configuration). It may also be possible to use the tri-box framework to describe more complex higher-order dependency, involving the creation or removal of script graph nodes under dependency control. To implement this would require a U-agent able to create entire tri-boxes somewhere in D . A dependency-driven parser would require such a U-agent.

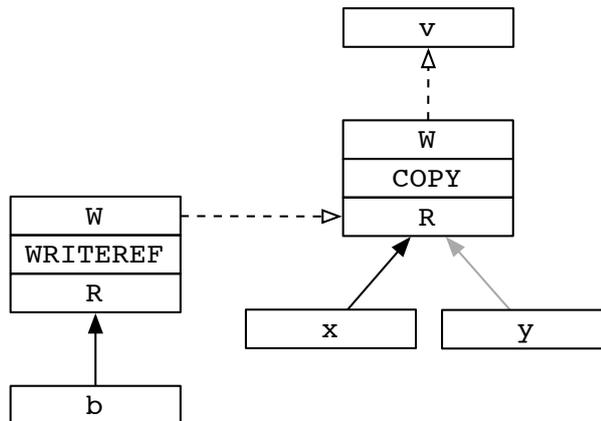


Figure 5.15: An `if` conditional HOD in the tri-box framework

Example 5: Subjective dependency

The characterisation of dependency from Beynon *et al* [BCSW99] quoted in §5.1.2 emphasises the *subjective* character of the dependency concept: “A dependency is a relationship between observables that pertains *in the view of a particular agent*” (my emphasis). Section §5.1.2 contains an example implemented in SR where two agents, O and O2, observe the three variables a, b and c. The agent O perceives the dependency relationship ‘a is b+c’ in the state, but the agent O2 perceives only unrelated variables. It is possible to describe such ‘subjective dependencies’ by adding observing agents to a tri-box diagram.

Figure 5.16 shows two O-agents observing five value boxes. The agent O1 perceives only the dependency described by the C:ADD tri-box, and the agent O2 only that described by the E:ADD tri-box. As a result, whenever the agent O1 makes an observation, the state will be consistent with the relationship ‘c is a+b’. However, the state may not be consistent with the relationship ‘e is c+d’ — the agent O1 is able to observe the state during the time period between the start of a change to the value of c and the end of the execution of the update operator E:ADD. Conversely, the agent O2 will always perceive state to be consistent with ‘e is c+d’, but not necessarily ‘c is a+b’.

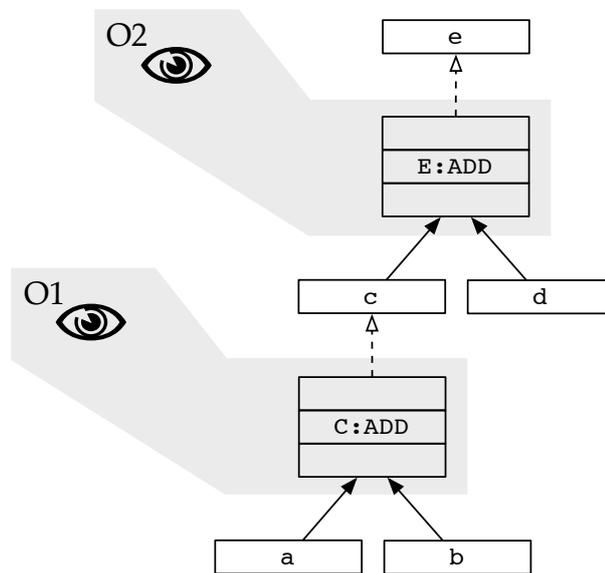


Figure 5.16: Two O-agents perceiving different dependencies

5.3.3 Implementation issues

The tri-box framework is a useful conceptual framework for understanding and describing the issues outlined in the previous section relating to the implementation of dependency. The framework provides a conceptual guide for implementation but it is not a detailed precise specification. Many questions remain to be answered before a full concurrent definition maintainer implementation can be considered. This section briefly outlines some of the issues for implementation that the framework raises by considering each example presented in the previous section in turn.

Example 1: The “power” script

A tri-box diagram can be implemented in many different ways. Each implementation requires decisions to be made about data structures and operational organisation.

The DAM machine data structure for the “power” script is shown in Figure 3.2 (p.115). In this implementation, two linked lists are attached to each value in the definitive store, one containing source and the other containing target pointers to other value locations. Note that in Figure 3.2, the Targets Store encodes the same information as the Sources Store. The two stores are used to improve the efficiency of change propagation, unlike the R- and W-sets of the tri-box framework, which do not hold redundant information.

A design using lists of source and target pointers works well if the number of arcs per script graph node is small. If this assumption is false (for example, if the source values for a definition are the entire screen state), then the lists may become unmanageably long.

In operation, use of the DAM machine involves the queuing of redefinitions which are then processed when an `update` routine is invoked. Considered at this level, the DAM machine is always in one of the two states — `QUEUING` or `UPDATING`. In terms of concurrency, the DAM machine therefore implements a ‘monolithic’ U-agent configuration, as shown in Figure 5.12. A concurrent definition maintainer design might implement a one-to-one U-agent configuration as shown in Figure 5.12.

One simple way to evaluate the design of a concurrent definition maintainer is to consider how it acts on two disjoint subgraphs of a script graph — for example, two copies of the “power” script. In a truly concurrent design, the synchronisation for

each subgraph should be independent: O-, C- and U-agents observing and acting on one subgraph should not be affected by other agents observing and acting on the other. Such a design is required to have no centralised data or entity which could reduce the independence of agents observing and acting upon subgraphs. This chapter has suggested that the primary parts of a concurrent definition maintainer are the data structure and protocols for concurrent interaction with that data structure — there can be no centralised single-threaded ‘definition maintainer’ code.

The “power” example illustrates the implications of the rule restricting the topology of a tri-box diagram: each box in S must be referenced by at most one W -set. Figure 5.12 (p.312) follows this rule faithfully. It should be noted that the restriction also applies to user input: in Figure 5.12, the user may not directly change the values contained in the value boxes e , f , g , h or i . We may assume that every box without an incoming W -arc implicitly has an incoming W -arc representing the agency of the user. The user may thus change the values contained in the ‘leaf’ value boxes a , b , c and d , and also the values contained in the tri-boxes. The restriction has much relevance to the topic of moding discussed in §5.2.2.

The final aspect of the “power” script to be raised here concerns the “power” function itself. Notice that, in the “power” script, `power()` is the only function which is not symmetric in its arguments (since $a + b = b + a$ etc, but $x^y \neq y^x$). In the one-to-one configuration in Figure 5.12, it is appropriate for every other tri-box to reference an R -set, but the $I:POWER$ update operator requires more information which could be described as an R -sequence. The tri-box framework is not based upon R - and W -sequences however — the framework is intended to describe the propagation of change and therefore sequencing of operator invocation. Sequencing of operator *arguments* is a detail that is not relevant at this level, and so the tri-box framework leaves this for the implementation. It is noteworthy here that the EDEN implementation creates a custom VM code ‘operator’ for each definition at parse time, encoding the references to operator arguments in the VM code. The EDEN scheduler then invokes the VM code at the scheduled time, without concern for the sequencing of operator arguments.

Example 2: Character glyphs

When discussing Example 1 above, we considered the scenario of a single definition having the entire screen state as its source value. Although this is possible in principle in the DAM machine, it would create an unmanageably long source pointer list and many short target pointer lists. Example 2 involves the dual situation, showing eight values (PW1...PW8) that each have a single source value (CODE). In the DAM machine, this aspect of Example 2 would require eight separate definitions, since in that design, operators may only return a single value. In contrast, in the tri-box framework, U-agent operators may write to many values. This example can be constructed as a single U-agent, or as eight U-agents (each writing to a single value), depending upon which is more appropriate.

In the tri-box framework, values are firstly separated from definitions, through the concepts of the ‘value box’ and the ‘tri-box’. Values and definitions are joined again through the use of the R- and W-set concepts, which, if *D-in-S* is countenanced, also allows the consideration of higher-order dependency, when the script graph information held in tri-boxes can also be used as value boxes.

The tri-box framework is more general than the DAM machine design, and also, in fact, more general than the concept of the LLDN upon which the DAM machine design is based. The tri-box framework may therefore have broad implications for designing and implementing definitive notations.

Example 3: Observation of overlapping subsets

The tri-box framework abandons objective symbolic reference in preference for R- and W-sets of references to individual locations. This conceptual step allows highly specific references to subsets of state to be constructed that different agents can use in a more ‘subjective’ manner, as shown in Example 3. However, if reference to subsets is implemented in its full generality, the implications are extremely costly.

If in implementation we wish an R-set to be able to refer to any possible subset of the state S (i.e. any possible element of the powerset $\mathcal{P}(S)$), then the simplest fixed-length representation is a bit-vector, with one bit per value location representing presence or absence of that location in the reference. Unfortunately, such a representation requires as many bits as there are locations: using a 32-bit word for

the representation only allows 32 locations. This problem is only partially redressed by the fact that, in the description of the framework above, we have not prescribed the length of the representation of a tri-box (although it is convenient to think of it conceptually as three boxes, it need not be represented in implementation as three value boxes).

However, in practice, it is unlikely that we will need to refer to every possible subset of S (i.e. every possible element of $\mathcal{P}(S)$). Many smaller (but less general) representations then become possible.

The DAM machine design implements a reference to a single location using a source pointer: a bit-vector interpreted as an address value. The source pointer is treated in the implementation as ‘opaque’ — it can only be used for dereferencing the particular location to which it points. Comparisons of source pointers or arithmetic on source pointers have no meaning — using terminology from the Java language, a DAM machine source pointer is a ‘reference’ rather than a ‘pointer’. Allderidge’s symbol table extension to the DAM machine in !Donald (see §3.3) persists in this design, as is conventional. Such a symbol table associates character string identifiers with values, but in the implementation, no significance is attached to structural relationships between identifiers, such as might stem (for example) from lexicographic ordering or spreadsheet cell naming conventions. It is therefore not usually meaningful to compare or perform arithmetic on character string identifiers.

The ‘opaque-ness’ of both of these forms of reference makes it necessary, when making a reference to multiple locations, to establish these by constructing lists of basic references (pointers or string identifiers).

A more general form of reference is provided in spreadsheet programs. For example, the “A1 reference style” in the spreadsheet program Excel (version ‘X’ for the Apple Macintosh) takes the following form (a regular expression synthesised from the documentation [exc01]):

$$\{ SN \{ : SN \} ! \} \{ CL \} \{ RN \} \{ : \{ CL \} \{ RN \} \}$$

where SN is a sheet name, CL is a column letter, RN is a row number, ‘!’ is the exclamation character, ‘:’ is the colon character, and curly brackets denote optional parts of the reference.

The colon delimits parts of the reference, some parts being optional. If *CLRN* is specified, then the reference denotes a single spreadsheet cell (e.g. B3). If all parts are specified, then each part of the reference specification can be thought of as a plane in three-dimensional space, the dimensions being columns, rows and ‘sheets’ of the current ‘workbook’ document. This reference representation (which has a fixed and small maximum length) therefore allows a reference to be made to any three-dimensional cube of cells (e.g. Sheet1:Sheet3!A5:C7).

Note however that it is not possible in the above syntax to reference a non-contiguous range of cells directly: although the formula =SUM(A1:B2,D1:E2) denotes the sum of the cells enclosed in the area A1:E2 but not C1:C2 (and therefore not a contiguous range), this formula is actually an invocation of the SUM() function with *two* reference arguments separated by a comma.

The tri-box framework assumes the most general form of reference possible, given a state *S* considered as a sequence of atomic ‘boxes’. The provision of less general, but still powerful, forms of reference appears to involve two ingredients: organising state ‘boxes’ into some kind of space, and having a means to form references into that space. Ascribing a suitable structure to the space greatly increases the power of references used in combination. The ‘suitability’ of the structure will depend upon the domain¹⁸.

Example 4: The if operator

The notion of *D-in-S* for higher-order dependency, made clear in the tri-box framework, is a conceptual breakthrough. Previously we have not been able to describe higher-order dependency in such concrete terms (see for example the discussion by Gehring *et al* in [GYC⁺96]). An ill-defined abstract research problem is thus transformed into a precise technical problem. The technical problems remaining to be solved include the following three issues:

- The tri-box script graph data must be accessed indivisibly by concurrent agents, as are the value boxes. Note that the SR code in Appendix §5.A (p.328) does not implement *D-in-S*: the semaphores in the code mediate access to value boxes only.

¹⁸*Cf.* the way in which spreadsheets are suited to financial and administrative applications where tabular data is common.

- Higher-order dependency may complicate the detection of script graph cycles.
- Levels of evaluation (or change propagation) may need to be implemented. For maximum efficiency in Figure 5.15 (p.318) for example, if \mathbf{b} and \mathbf{x} are simultaneously changed, the `WRITEREF` operation must take place before the `COPY` operation.

Example 5: Subjective dependency

P-H. Sun first experimented with what is here termed ‘subjective dependency’ in his `dtkeden` extension to `tkeden` [Sun99] (see also §4.1.6), where `dtkeden` clients interact over a network via a `dtkeden` server, communicating by sending redefinition strings through TCP/IP sockets. Clients may have private local state and can maintain their own private dependencies between public data.

The tri-box framework lends clarity to what it means for two agents to have observables in common but to perceive different dependencies amongst them. In Figure 5.16 (p.320), each of the agents O1 and O2 has a different D . In this example, the two D s are disjoint, but examples with some commonality between agent D s are easily envisaged.

Implementing subjective dependency poses problems of distinguishing tri-boxes from value boxes and associating tri-boxes with agents. The association is particularly (perhaps only) important when synchronising agent action and observation, so one implementation design would be for an agent to identify its particular D when a protocol is used. The tri-box conceptual framework seems general enough to describe implementations ranging from distributed (with the state S distributed amongst multiple computers and the protocols implemented using message passing) to shared-memory (with the state S existing in the shared memory and protocols implemented using atomic processor instructions) to single-processor (with the state S existing in the single memory and a scheduler determining which agent to animate next on the basis of information resulting from protocol calls).

The diversity of the implementation issues raised by the above five examples of applying the tri-box framework suggests that no single ideal general purpose definition maintainer implementation exists. There may be an ideal if we restrict our usage and expectations of the tool to the types of Empirical Modelling performed

with `tkeden` before 1999. However, new developments in applying EM, such as adding 3D graphical realisation and real-time input, lead directly to problems of organising the state space and concurrent real-time issues to which the best solution will depend both upon the requirement and the hardware available.

5.A Concurrent definition maintenance in SR

```

2 # A concurrent agent-based definition maintainer case study
# Ashley Ward (ashley@dcs.warwick.ac.uk)
# June-July 2003
4
resource agentdep()
6
  type debugprint = string[6]
8
  # upper bound on dependency table index. Table is indexed 1:ND
10  const ND := 5
12
  # upper bound on value store. Store is indexed 1:NS
  const NS := 9
14
  type dref = int # 1:ND
16  type sref = int # 1:NS
18
  type drefset = [ND] bool
  type srefset = [NS] bool
20  type refset = [*] bool
22
  const EMPTYDREFSET := ([ND] false)
  const EMPTYSREFSET := ([NS] false)
24  const FULLDREFSET := ([ND] true)
26
  # the store of values
  var s[NS]: int
28
  # the dependency table. The order of arguments is sometimes
30  # important (eg subtraction), but the table does not encode
  # each dependency precisely, just the info needed to describe the
32  # dependency tree. The exact implementation of each dependency is
  # described within the relevant u.
34
  var o[ND]: srefset := ([ND] EMPTYSREFSET)
36  var i[ND]: srefset := ([ND] EMPTYSREFSET)
38
  type opid = enum(ADD, TIMES, MAX3, POWER)
  var u[ND]: opid
40  op update(dref; debugprint) {send, call}
42
  sem l[ND] := ([ND] 1)
44
  var undef[ND]: bool := FULLDREFSET
46
  # end of dependency table
48
  type lockopt = enum(LOCK, UNLOCK)
  type recurseopt = enum(RECURSE, NORECURSE)
50

```

```
52  /*
    * refset functions
    */
54  /*
56  procedure nill()
58  end
    */
60  # bit-wise OR two drefsets together
62  procedure ordrefset(a: drefset; b: drefset) returns aorb: drefset
    var i: int
64
    aorb := EMPTYDREFSET
66
    /*
68  co (i := 1 to ND st a[i] or b[i]) nill() -> aorb[i] := true oc
    */
70
    fa i := 1 to ND ->
72     if a[i] or b[i] -> aorb[i] := true; fi
    af
74
    end
76
    # bit-wise OR two srefsets together
78  procedure orsrefset(a: srefset; b: srefset) returns aorb: srefset
    var i: int
80
    aorb := EMPTYSREFSET
82
    fa i := 1 to NS ->
84     if a[i] or b[i] -> aorb[i] := true; fi
    af
86
    end
88
    # bit-wise AND two srefsets together
90  procedure andsrefset(a: srefset; b: srefset) returns aandb: srefset
    var i: int
92
    aandb := EMPTYSREFSET
94
    fa i := 1 to NS ->
96     if a[i] and b[i] -> aandb[i] := true; fi
    af
98
    end
100
```

```
102 # is the given refset empty (ie all false)
103 # or does it contain a true value?
104 procedure isemptyrefset(r: refset) returns empty: bool
105     var i: int
106
107     empty := true
108
109     fa i := 1 to ub(r) ->
110         if r[i] ->
111             empty := false
112             exit
113         fi
114     af
115 end
116
117 # convert a refset to a string for debug printing purposes
118 procedure refsettostring(r: refset) returns s: string[80]
119     var i: int
120
121     s := ""
122
123     fa i := 1 to ub(r) ->
124         if r[i] ->
125             s := s || string(i)
126         fi
127     af
128 end
129
130
131
132
```

```
134  /*
135   * procedures to manipulate the dependency table
136   */
137
138  # recursively all sources (D space) of all r (S space), including r
139  procedure dsources(r: srefset) returns ret: drefset
140     var d: dref
141
142     ret := EMPTYDREFSET
143
144     fa d := 1 to ND ->
145         if not isemptyrefset(andsrefset(o[d], r)) ->
146             ret[d] := true
147             ret := ordrefset(ret, dsources(i[d]))
148         fi
149     af
150
151 end
152
153 # recursively? all targets (D space) of all r (S space), excluding r
154 procedure dtargets(r: srefset;
155     recurse: recurseopt) returns ret: drefset
156     var d: dref
157     var ts: srefset
158
159     ret := EMPTYDREFSET
160
161     fa d := 1 to ND ->
162         if not isemptyrefset(andsrefset(i[d], r)) ->
163             ret[d] := true
164             if recurse = RECURSE ->
165                 ret := ordrefset(ret, dtargets(o[d], recurse))
166             fi
167         fi
168     af
169
170 end
```

```

# lock/unlock this set of Ds
172 procedure lockset(lock: lockopt; locks: drefset; dp: debugprint)
    var d: dref
174
    if lock = LOCK ->
176       # as long as concurrent competing processes make their locks in the
          # same sequential order, deadlocks are prevented
178
          fa d := 1 to ND ->
180             if locks[d] ->
                  write(dp, "P(", d, ")")
182                 P(l[d])
                  fi
184             af

186     [] lock = UNLOCK ->
          /* co (d := 1 to ND st locks[d]) V(l[d]) oc */
188
          fa d := 1 to ND ->
190             if locks[d] ->
                  write(dp, "V(", d, ")")
192                 V(l[d])
                  fi
194             af

196     fi

198 end

# are any of these s marked as undefined in the dependency table?
200 procedure containsundef(sources: srefset) returns containsundef: bool
202     var sr: sref
    var dr: dref
204
    containsundef := false
206
    fa dr := 1 to ND ->
208       if not isemptyrefset(andsrefset(o[dr], sources)) ->
          if undef[dr] ->
210             containsundef := true
                exit
212             fi
          fi
214       af

216 end

218

```

```

220  /*
      * observe / change protocol procedures
222  */

224  # prohibit change (& observation?) of all of r and sources
      # so that state appears consistent
226  procedure preOlock(r: srefset; dp: debugprint) returns locked: drefset
      # lock any use of any r and all sources down the tree
228      locked := dsources(r)

230      lockset(LOCK, locked, dp)

232  end

234  # allow change of previously locked dependencies
      op postOunlock(drefset; debugprint) {send}
236  proc postOunlock(locked, dp)
      lockset(UNLOCK, locked, dp)
238
      end

240
242  # prohibit observation (& change?) of all of r and targets
      procedure preCLOCK(r: srefset; dp: debugprint) returns toupdate: drefset
      var targets: drefset

244
      # lock any targets of all r and all targets up the tree (excluding r)
246      targets := dtargets(r, RECURSE)

248      lockset(LOCK, targets, dp)

250      # now mark all targets (excluding r) up the tree as undefined
      undef := ordrefset(undef, targets)
252
      # the first level of targets of r should now be updated
254      # (and recursively, the targets of those targets)
      toupdate := dtargets(r, NORECURSE)
256
      end

258
260  # update first level of dependencies, allow observation/change, then
      # propagate dependency update up to the next level
      op postCupdateunlock(drefset; debugprint) {send, call}
262  proc postCupdateunlock(toupdate, dp)
      var d: dref

264
      # in parallel, invoke the necessary update procs, which should each
266      # read their input, write output, unlock, then invoke the next level
      # of dependency update
268      co (d := 1 to ND st toupdate[d]) call update(d, dp) oc

270  end

```

```

272
274 /*
276  * update dependency procedures
278 */
280 # dirty hack for getting the refno'th single reference out of a set
282 procedure oneref(set: srefset; refno: int) returns s: sref
284   var i: sref
286   s := 0
288   fa i := 1 to NS ->
290     if set[i] and (--refno = 0) ->
292       s := i
294       exit
296     fi
298   af
300 end
302 # update the s outputs of a dependency. This is called from update and
304 # also possibly manually when a dependency is changed
306 procedure valueupdate(d: dref)
308   if u[d] = ADD ->
310     s[oneref(o[d], 1)] := s[oneref(i[d], 1)] + s[oneref(i[d], 2)]
312   [] u[d] = TIMES ->
314     s[oneref(o[d], 1)] := s[oneref(i[d], 1)] * s[oneref(i[d], 2)]
316   [] u[d] = MAX3 ->
318     s[oneref(o[d], 1)] := max(s[oneref(i[d], 1)],
320                               s[oneref(i[d], 2)],
322                               s[oneref(i[d], 3)])
324   [] u[d] = POWER ->
326     s[oneref(o[d], 1)] := s[oneref(i[d], 1)] ** s[oneref(i[d], 2)]
328   fi
330 end

```

```
310 # update state for a particular dependency by reading input and writing
311 # output, then unlock the dependency and invoke the next level of
312 # dependency update
proc update(d, dp)
314   var args: int
315   var sr: sref
316
317   write(dp, "UPDATE", d)
318
319   if containsundef(i[d]) ->
320     # at least one source value is undefined: ignore now,
321     # don't propagate change upwards and wait for the final update call
322     write(dp, "UPDATE undefined source")
323     return
324   fi
325
326   if not undef[d] ->
327     # this d has already been updated
328     write(dp, "UPDATE value already defined")
329     return
330   fi
331
332   valueupdate(d)
333
334   # the output value is no longer undefined
335   undef[d] := false
336
337   write(dp, "V(", d, ")")
338   V(l[d])
339
340   call postCupdateunlock(dtargets(o[d], NORECURSE), dp)
341
342 end
343
344
```

```
346  /*
347  * debug test code
348  */
349
350  procedure writesd()
351  var sr: sref
352  var dri: dref, dr: dref
353
354  fa sr := 1 to NS ->
355
356      # find this s in D space
357      dr := 0
358      fa dri := 1 to ND ->
359          if o[dri][sr] ->
360              dr := dri
361              exit
362          fi
363      af
364
365      writes(" S:", sr, " = ", s[sr])
366
367      if dr != 0 ->
368          writes("\tD:", dr,
369              " o:", refsettostring(o[dr]),
370              " i:", refsettostring(i[dr]),
371              " u:", u[dr],
372              " l:?", # can't print l[dr]
373              " undef:", undef[dr],
374              "\n")
375      [] else ->
376          write()
377      fi
378
379  af
380
381  end
382
```

```

384 # "Power script" - example from Cartwright p. 123
386 o[1] := ([6] false, true, [2] false); # g
388 i[1] := ([4] false, true, true, [3] false); # e, f
388 u[1] := TIMES;

390 o[2] := ([8] false, true); # i
392 i[2] := (true, [6] false, true, false); # a, h
392 u[2] := POWER;

394 o[3] := ([7] false, true, false); # h
396 i[3] := ([3] false, true, false, true, true, [2] false); # g, f, d
396 u[3] := MAX3;

398 o[4] := ([4] false, true, [4] false); # e
400 i[4] := (true, true, [7] false); # a, b
400 u[4] := ADD;

402 o[5] := ([5] false, true, [3] false); # f
404 i[5] := (false, true, true, [6] false); # b, c
404 u[5] := ADD;

406 # initialise values but not dependencies
408 s := (1, 2, 3, 4, [5] -1)
408 undef := FULLDREFSET

410 writesd()

412 # simulate change to all non-d to initialise. starting with just a
414 # (1) or c (3) won't work as then e (4) or f (5) will be undefined.
414 var sr: srefset
416 sr := ([4] true, [5] false) # a,b,c,d

416 var toupdate: drefset
418 toupdate := preClock(sr, "INIT")

420 write("TOUPDATE", refsettostring(toupdate))

422 postCupdateunlock(toupdate, "INIT")

424 writesd()

426 # observe g
428 sr := EMPTYSREFSET
428 sr[7] := true # g
430 var locked: drefset
430 locked := preOlock(sr, "OG")
432 write("LOCKED", refsettostring(locked))
432 write("S7=", s[7])
434 send postOunlock(locked, "OG")

```

```
436  /*
438  * Concurrent test processes
438  */
440  # observe a single s value. Perhaps not useful as no possibility for
440  # simultaneity of observation.
442  procedure Os(sr: sref; dp: debugprint) returns v: int
444    var sset: srefset
444    var locked: drefset
446
446    sset := EMPTY_SREFSET
446    sset[sr] := true
448
448    locked := preOlock(sset, dp)
450    v := s[sr]
450    send postOunlock(locked, dp)
452
452  end
454
454  # change a single s value (not a d)
456  procedure Cs(sr: sref; v: int; dp: debugprint)
458    var sset: srefset
458    var toupdate: drefset
460
460    sset := EMPTY_SREFSET
460    sset[sr] := true
462
462    toupdate := preCclock(sset, dp)
464    s[sr] := v
464    send postCupdateunlock(toupdate, dp)
466
466  end
468
```

```
470 /*
471 # simple observation of just one value at a time, observing dependency
472 # constraint (although if there is no simultaneity, this is
473 # questionable)
474 process 01
475     var i: int
476
477     fa i := 1 to 20 ->
478         nap(int(random(100)))
479         write("01 S7", 0s(7))
480
481     af
482 end
483
484 # simple observation of just one value at a time
485 process 02
486     var i: int
487
488     fa i := 1 to 20 ->
489         nap(int(random(100)))
490         write("02 S7", s[7])
491
492     af
493 end
494 */
```

```

# Od observes abcefg simultaneously and perceives the dependencies
496 process Od
    var i: int
498     var sset: srefset
        var locked: drefset
500     var dp: debugprint := "  Od"

502     fa i := 1 to 20 ->
        nap(int(random(100)))
504
        sset := ([3] true, false, [3] true, [2] false) # abcefg
506
        locked := preOlock(sset, dp)
508     writes(dp, ": S1=", s[1], " S2=", s[2], " S3=", s[3],
        " S5(1+?*2)=", s[5], "(" , s[5]=s[1]+s[2], ") ",
510        " S6(2+3)=", s[6], "(" , s[6]=s[2]+s[3], ") ",
        " S7(5*6)=", s[7], "(" ,s[7]=s[5]*s[6], ") \n")
512     send postOunlock(locked, dp)

514     af

516 end

# On observes abcefg simultaneously, but does not perceive dependency
518 process On
    var i: int
520     var dp: debugprint := "  On"

522
524     fa i := 1 to 20 ->
        nap(int(random(100)))

526     writes(dp, ": S1=", s[1], " S2=", s[2], " S3=", s[3],
        " S5(1+?*2)=", s[5], "(" , s[5]=s[1]+s[2], ") ",
528        " S6(2+3)=", s[6], "(" , s[6]=s[2]+s[3], ") ",
        " S7(5*6)=", s[7], "(" ,s[7]=s[5]*s[6], ") \n")

530     af

532 end
534

```

```
536 # C1 changes the value of S1 (a), observing dependency action
537 # constraints
538 process C1
539   var i: int
540   var randomv: int
541   var dp: debugprint := "C1"
542
543   fa i := 1 to 20 ->
544     nap(int(random(100)))
545
546     randomv := int(random(10))
547     write("C1: S1=", randomv, "...")
548     Cs(1, randomv, dp)
549   af
550
551 end
552
553 # C2 changes the value of S2 (b), observing dependency action
554 # constraints
555 process C2
556   var i: int
557   var randomv: int
558   var dp: debugprint := " C2"
559
560   fa i := 1 to 20 ->
561     nap(int(random(100)))
562
563     randomv := int(random(10))
564     write("C2: S2=", randomv, "...")
565     Cs(2, randomv, dp)
566   af
567
568 end
```

```
570 # C3 changes D4 (value at S5) between TIMES and ADD, observing
# dependency action constraints
process C3
572   var i: int
   var sset: srefset
574   var toupdate: drefset
   var dp: debugprint := " C3"
576
   # whether D4 is TIMES or ADD (note operators chosen such that errors
578   # cannot occur with source values of 0)
   var t: bool := true
580
   fa i := 1 to 20 ->
582     nap(int(random(100)))
584
     sset := EMPTY_SREFSET
     sset[5] := true # e
586
     toupdate := preClock(sset, dp)
588
     writes("C3: S5 becomes ")
     if t ->
590       write("TIMES")
       u[4] := TIMES # note dref, not sref
592     [] else ->
       write("ADD")
       u[4] := ADD
594     fi
596
     # have to recalculate the s value manually for a d change
     write(dp, "VALUEUPDATE 4")
     valueupdate(4)
600
     send postCupdateunlock(toupdate, dp)
602
     t := not t
604
   af
   end
606
608 end
```

Chapter 6

Conclusions, contributions and future work

This thesis has made a wide variety of contributions which can be categorised for the purposes of this conclusion into the following four themes:

- Conceptual machine models for EM;
- Empirical study of the EM tools and background scholarship;
- Identification of subtleties associated with dependency, and
- Strengthening connections with other work.

The next four subsections elaborate on the contributions made by this thesis to each of these themes. The final section hints at future application for this work.

6.1 Conceptual machine models for EM

Conceptual machine models of the ADM, the DAM machine and EDEN have been closely examined and related for the first time. This has involved significant original research and analysis as well as the refinement of pre-existing work. I have proposed a framework within which definitive systems implementations can be usefully understood. This framework is based on the ideas of Slade and encompasses a range of evaluation/storage strategies.

Slade's Abstract Definitive Machine (ADM) [Sla90] exemplifies a machine model that can be implemented using an "evaluate-at-use" strategy. This strategy is relatively simple to implement as only formulae need be stored. The emphasis is then on *actions* that cause evaluation, and dependency has a relatively low profile. I highlighted some ambiguities in the model of ADM execution as it has been described in the EM literature, introducing the concepts of major and minor state transitions in order to explain the ambiguity. These concepts are then used in later chapters.

Cartwright's Definitive Assembly Maintainer (DAM) machine [Car99] exemplifies an alternative "evaluate-at-redefinition" strategy. This strategy requires that values be stored in addition to formulae. The focus is then on *dependency*, which prescribes how change resulting from redefinitions should be propagated through the stored values in order to update the state. I highlighted some deficiencies in the DAM machine. Cartwright's Dependency Maintainer Model (DMM), on which the DAM machine implementation is based, formalises definitive notations as they currently exist. I make this more explicit by showing in greater detail the Low Level Definitive Notation (LLDN) concept assumed by Cartwright's DMM. My tri-box model presented in §5.3 is a proposal for a richer machine model to support dependency maintenance beyond what is possible with our current definitive notations. I have described the model using graphical examples. Questions remain about how to represent this model in a notation.

All other strategies for dependency maintenance can be regarded as blending evaluation-at-use with evaluation-at-redefinition. Such a strategy is illustrated in Y.W. Yung's EDEN [Yun90] which uses an "evaluate-at-use-when-necessary" strategy. The Eden language and EDEN implementation successfully integrate both actions *and* dependency, and, as a result, a wide variety of models can be constructed

using it. This thesis includes the first full exposition of the EDEN machine model, developed by deriving a pseudo-code for the scheduler algorithm and representing it in two types of diagram. This reveals many clever aspects of Yung’s design. My exposition suggests ways of implementing higher order dependency using the existing Eden language, with a higher degree of confidence in its correctness. It also emerges that the EDEN algorithm also achieves an aim set out by Cartwright in [Car99]: that of minimising updates resulting from a block redefinition.

Despite these many positive aspects, the EDEN design is complex and sequential and it is difficult to see how to extend it to take advantage of the possibilities for concurrent update of dependencies. However, the integration of actions and dependency as distinct concepts within the Eden language allows me to show how definitions can be translated to actions, thereby making my notion of ‘dependency-as-agency’ more concrete. It then becomes clear that definitions are a form of triggered action with certain limitations. I name these actions *definition-agents*. This establishes the context for an initial analysis of concurrent definition maintenance in Chapter 5. The two primary issues are how to map definition-agents to the script graph and then how to synchronise (or sequentially, schedule) the actions of the definition-agents.

6.2 Empirical study of the EM tools and background scholarship

Building on some preliminary tests made by Cartwright, I compared the DAM machine with EDEN, paying particular attention to controlling differences of underlying platform. The tests show that the two definition maintainers actually have similar performance: any advantages that the !Donald application has over `tkeden` appears to be attributable to the different graphics subsystems used, not to the definition maintainer implementation.

Considering possible ways to evaluate a script graph, I constructed some minimal examples that can be used in black-box testing of any definition maintainer

in order to investigate the evaluation ordering that it implements. An analysis of the schedule ordering used by EDEN reveals that it evaluates breadth-first, oldest-to-newest. Definitions and actions appear to be scheduled using the same strategy. This observation feeds into the discussion of the distinction between dependency and agency mentioned above.

In various attempts to quantify aspects of the context of EDEN, I collected historical source code and analysed code growth. The resulting data shows continued growth since Y.W. Yung's original version of 1988, and also illustrates an early decision that I took as the maintainer of EDEN: to attempt to grow the tool through writing code in Eden rather than lower-level languages, in order to assist with portability, reliability and flexibility. I also compiled a table summarising over two years of data that I collected in order to measure usage of the tool. In the thesis, this illustrates the pattern of recent local usage.

These quantified aspects of historical context are mixed with careful reading of the various original sources, which have been reviewed in the light of our current understanding. Drawing together the writings exposes various inconsistencies due to the immaturity of our research area. In particular, there is a problem with the various terminologies used by different authors. The variation comes about for many reasons. It is partly due to continuing improvements in our understanding of the various topics. The broad scope of EM also causes difficulties, particularly for authors investigating applications of EM, who then have to deal with terminology both from EM and from their application area. This seems to have caused the meaning of some terms relating to EM machine models in particular to be inadvertently broadened beyond their original scope. I also suspect that limited accessibility of the original source material has made consistency difficult, and in order to conduct some of the research necessary for this thesis I have first made various efforts to improve the organisation and availability of our library. In this thesis, I have attempted to point out problems in some of the sources in particular in relation to machine models, and efforts are underway more generally within the group to define standard terms where we have developed them and to encourage their use.

6.3 Identification of subtleties associated with dependency

My introduction attempts to show the range of applicability of the dependency concept. An appreciation of the concept in full inevitably involves the intimately related concepts of observables and agency. Even if we restrict our attention to the domain of programming, various subtleties associated with dependency exist, which I attempt to highlight in this thesis.

Some of the issues have been identified before but have not received much attention. Meziani's [Mez87] concept of 'moding' for example is one of the early contributions that has had no effect on subsequent tool building. However it is an important concept that has significant bearing on problems we still encounter with composite structures such as definitive lists. I have reviewed moding problems in relation to broader problems of dependency maintenance for which this thesis offers partial solutions.

Some of these problems are considered in full for the first time in this thesis. The indirection involved in a symbolic reference is a problem that I have illustrated in the context of the DAM machine, where it is most clearly shown. I have also encountered the same problem in respect of several dependency structures that can be implemented only with difficulty in the current EDEN. The simplest problematic example illustrated in this thesis is that of the 'if' Higher-Order Dependency (HOD). This is a particular problem for implementations that use doubly-linked pointers to indicate sources and triggers of symbols, as both EDEN and the DAM machine do. Implementations that use the evaluate-at-use strategy avoid this problem, as the focus is on actions rather than propagation of change. Since the essence of the 'if' HOD problem lies in the detection and propagation of change rather than in evaluation, this problem has not been previously encountered in the Computer Science literature, where the emphasis is on implementing sequences of actions. The implementers of spreadsheet programs face the same problem with the INDIRECT function, and usually work around the problem rather than solving it in full. I have proposed a "tri-box model" that takes full account of the problem and provides a potential solution, but it may be hard to implement the model efficiently as it is currently conceived.

6.4 Strengthening connections with other work

Although the main body of this thesis contains only a few references to external literature, the thesis makes new and significant connections with other work. Technically, this is the first work to give a substantial treatment of the issue of concurrent dependency maintenance. Dependency requires a form of synchronisation that is reconfigured in varied ways depending upon factors including the topology of the current script graph and the location of change within it. A definition can be considered to be a new kind of synchronisation primitive that has interesting and useful properties: it gives a guarantee about the validity and explanation of the current value without the need for information hiding. Definitive scripts can be composed with other scripts, and can be connected to other procedural systems if sufficient knowledge of change is available.

In respect of the underlying philosophy for Empirical Modelling (that of generating “one experience that knows another”), this thesis emphasises the importance of tool building. Our concepts have been progressively refined and tested in application. However, the tools do not implement the concepts faithfully in many cases. In particular, I take the view in this thesis that our implementation techniques for tools should *embody* the concepts at as low a level as possible, rather than take the approach that the concepts are something to be implemented through abstraction. A computer tool that embodies the principles will provide a substrate to a modeller that can be compared to the pipes, tanks and fluid used by Phillips in constructing his machine (see §1.1). Such a substrate allows a modeller to interact with state in ways that are in no respect preconceived. A model constructed in such a substrate can then potentially be an extremely powerful metaphor for the modeller’s understanding, giving the fullest possible support for interaction with meaningful state.

My experiments with DAM machine dependent pixels, the EDEN steering wheel and railway are the first to extend our concepts to broader aspects of computer-related technology. They demonstrate the more comprehensive view of computing which we intend EM to embrace.

6.5 Future work: dependable computing, psychology and foundations

I believe that dependency can in principle support more dependable computing. The prospects here are highlighted by the following quote from McCormick¹, which is also cited in the introduction.

[Software] developers have always had to explain relationships within and between their systems. If they can explain those relationships with the simplicity and consistency demanded of other engineering disciplines, they will succeed. If not, it probably means that a dash for novelty has sprinted too far, too fast, and too soon.

Section §1.4 in the introduction looked briefly at circuits constructed from digital logic gates. They can be described in an uncontroversial diagrammatic form and easily extended or composed with other circuits. Such circuits can be specified using the simple and powerful Boolean algebra. These are illustrative of the simple, consistent, explicable relationships desired of software by McCormick.

It seems to me that our current foundations for computing make it exceptionally hard to construct programs that exhibit the simple, consistent and explicable relationships we require for dependable computing. This thesis contends that many of the problems of dependable computing stem from the pervasive reliance on sequences of action in the conception of software. It further proposes that indivisibility in state change — the central abstraction underlying circuits — can be the basis of a radical generalisation of circuit building.

Two aspects of circuits are particularly relevant to the research in this context:

- it is possible to exploit circuits in a wide range of applications (albeit sometimes courtesy of the von Neumann machine);
- circuits are rooted in a mature engineering discipline that provides guarantees about the simplicity and consistency of the relationships they establish.

Extensive previous work on Empirical Modelling has indicated that “generalised circuits” can potentially be exploited in just as wide a range of applications and that, moreover, it can address issues that are treated as separate concerns in the software-as-sequences-of-actions paradigm. The main contribution of this thesis is

¹[McC] unpublished essay, quoted in [Lev95, p.509].

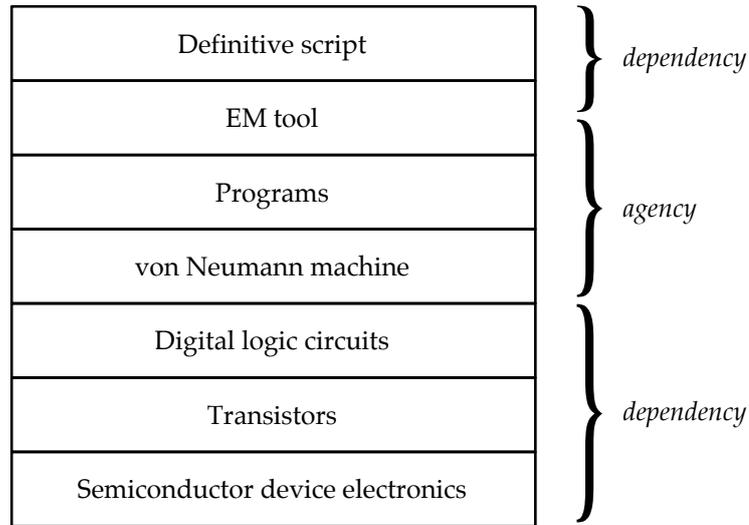


Figure 6.1: Dependency and agency in the current EM tool architecture

to explore the extent to which such generalised circuits can be rooted in a mature engineering discipline. In particular, the critiques of machine models show how the implementation and use of dependency may be amenable to engineering analysis.

There are two principal ways in which the research reported in this thesis can be extended towards the goal of supporting more dependable computing. Figure 6.1 depicts the way in which dependency in software and in circuits are related in the architecture of current EM tools. In the short term, it will be important to further the study of dependency-as-agency that was initiated in the analysis of EDEN in Chapter 4 and the DAM machine in Chapter 3. In the longer term, it may be possible to reduce the contribution that agency makes to the architecture. This contribution is already being eroded by the progressive re-implementation of EDEN within itself alluded to in Chapter 4. More radical ways of eliminating agency are suggested by the direct exploitation of the DAM machine video hardware described in §3.5.4.

But how will we know if we have succeeded in offering support for more dependable computing? We must move our studies beyond the anecdotal, perhaps first to case studies showing a correlation between the use of dependency and a more dependable product, and then ideally to a full true experiment demonstrating

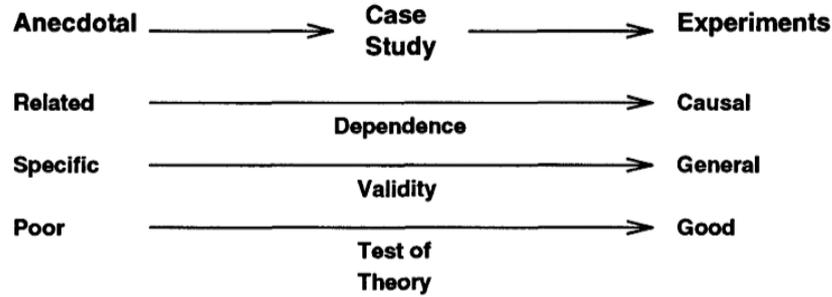


Figure 6.2: Spectrum of empirical work (from [VP95], © 1995 ACM — copy by permission of the Association for Computing Machinery)

causality and a constructive, testable theory [VP95] that explains why dependency enables us to produce a more dependable product. Figure 6.2 illustrates this range of possible empirical work.

It may be possible to construct such experiments in the classroom. In 2003-04, we set an assignment for 16 fourth year Computer Science MEng students taking our “Introduction to Empirical Modelling” module, firstly asking each of them to independently produce a `tkeden` model of a home central heating system. In a second (initially unseen) part to the assignment, we asked them to change their model in order to demonstrate scenarios involving heating system maintenance, malfunction and design changes (e.g. model the temporary disconnection of a radiator, a leaking boiler, the addition of a frost-protection feature). Our hypothesis is that a model which is largely constructed using dependency and based on an LSD account, as we recommend, should be easier to modify to these changed requirements than a simulation programmed largely from sequences of procedural actions without close and continuous connection of meaning to the referent. The assignment submissions taken as a whole appear to confirm this — there appears informally to be a correlation between the use of dependency and the ease of making changes in the second part of the assignment, but the data does not support a formal evaluation. Therefore we are still at the level of anecdotal evidence. Possibly a controlled experiment where one group of subjects would use the definitive and procedural features of EDEN, and another group only the procedural side, would be a good basis for such an evaluation. Continuing our exploration of the Cognitive Dimensions of definitive

notations, which we started in [BRWW01], also seems likely to lead to insights.

As with any contribution to human-computer interaction, it is necessary to produce hard, quantitative results, so that hypotheses may be confirmed or falsified. However, it seems that proper experimental design, analysis and reporting (by the standards of the psychology discipline) is still rare in the Computer Science field. Sheil [She81] states that:

As practiced by computer science, the study of programming is an unholy mixture of mathematics (e.g. [Dij76]), literary criticism (e.g. [KP74]) and folklore (e.g. [FPB95]).

and recommends that:

... the methodological recommendations of computer science should be recognised as *empirically testable, psychological hypotheses*... a discipline of computer science has an obligation to validate these claims.

(It should be noted that Empirical Modelling is not a method, rather an “amethodical approach” as described by [TBT00], although it may be a methodology in the sense described by [Che99].)

There are many risks to the validity of experiments such as ours — one example is the effect of “demand characteristics” mentioned by Sheil, whereby participants modify their behaviour in response to their perception of the experimenter’s desires. More recently, Kitchenham *et al* [KPP⁺02] state that in their view, “the standard of empirical software engineering research is poor”, but go on to present what appear to be useful guidelines for such research.

This thesis has been an attempt to analyse and ground the concept of dependency and its implementation. The most formal description we now have is given by Chapter 5, which relates the concept to observation and agency. The ACM Task Force on the Core of Computer Science, chaired by Denning, reported [DCG⁺89] in 1989 that “The fundamental question underlying all of computing is, What can be (efficiently) automated[?]”. The concept of dependency developed in this thesis is one that supports the indivisible change of meaningful state. To this extent, dependency offers an important and fundamental means of achieving automation. If programming — in a broad sense — refers to how all computer-based automation is to be managed, then dependency must play an essential role, complementary to that of logic, in the foundations of programming.

Bibliography

- [ABCK⁺90] W. Aspray, A.G. Bromley, M. Campbell-Kelly, P.E. Ceruzzi, and M.R. Williams. *Computing before Computers*. Iowa State University Press, 1990. (Cited on page 12.)
- [ABCY98] J.A. Allderidge, W.M. Beynon, R.I. Cartwright, and Y.P. Yung. Enabling technologies for empirical modelling in graphics. In *Proc. Eurographics UK, 16th annual conference*, pages 199–213, 1998. (048). (Cited on pages 98, 124 and 156.)
- [ABH02] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 247–259. ACM Press, 2002. (Cited on page 40.)
- [acm] ACM Digital Library. ACM, Inc. <http://portal.acm.org/> (accessed August 2004). (Cited on page 34.)
- [aco88] *BBC BASIC guide*. The Archimedes Series. Acorn Computers Limited, first edition, 1988. (Cited on page 149.)
- [aco92] *RISC OS 3 Programmer's Reference Manual*. Acorn Computers Limited, first edition, December 1992. (Cited on pages 110 and 111.)
- [AEWJ⁺02] A. Allan, D. Edenfeld, Jr. W.H. Joyner, A.B. Kahng, M. Rodgers, and Y. Zorian. 2001 technology roadmap for semiconductors. *Computer*, 35(1):42–53, January 2002. (Cited on page 220.)
- [age] AgentSheets, Inc. <http://agentsheets.com/> (accessed August 2004). (Cited on page 37.)
- [All97] J.A. Allderidge. Applying the Definitive Assembler Maintainer (DAM) architecture to graphics. Final year undergraduate project report May, University of Warwick Department of Computer Science, 1997. (Cited on pages 98, 124, 129 and 137.)
- [Ams86] Jonathan Amsterdam. Programming project: build a spreadsheet program. *BYTE*, 11(7):96–108, 1986. (Cited on page 33.)

-
- [ant] Apache Ant <http://ant.apache.org/> (accessed August 2004). (Cited on page 39.)
- [AO93] Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language*. Benjamin/Cummings, 309 Bridge Parkway, Redwood City, CA 94065, 1993. (Cited on page 289.)
- [app87] *Human Interface Guidelines: The Apple Desktop Interface*. Addison-Wesley, 1987. (Cited on pages 11 and 247.)
- [AS83] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Comput. Surv.*, 15(1):3–43, 1983. (Cited on page 24.)
- [Baa88] Erik H. Baalbergen. Design and implementation of parallel make. *Computing Systems*, 1(2):135–158, 1988. (Cited on page 39.)
- [BABH86] W.M. Beynon, D. Angier, T. Bissell, and S. Hunt. DoNaLD: A line-drawing system based on definitive principles. Technical report, University of Warwick Department of Computer Science, 1986. (Cited on pages xiv, 127 and 209.)
- [Bac78] John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978. (Cited on pages 29 and 63.)
- [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 Cooperative working in Concurrent Engineering*, pages Digest 1994/177, 9/1–9/4, October 1994. (040). (Cited on pages 77, 78 and 89.)
- [BACY94b] W.M. Beynon, V.D. Adzhiev, A.J. Cartwright, and Y.P. Yung. A computational model for multiagent interaction in concurrent engineering. In *Proc. CEEDA'94, Bournemouth University*, pages 227–232, 1994. (034). (Cited on pages 77, 78 and 89.)
- [BACY94c] W.M. Beynon, V.D. Adzhiev, A.J. Cartwright, and Y.P. Yung. A new computer-based tool for conceptual design. In *Proc. Workshop Computer Tools for Conceptual Design*, pages 171–188, 1994. (035). (Cited on page 89.)
- [BAD⁺01] M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Reichwein, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11(2):155–206, March 2001. (Cited on pages 37 and 249.)
- [Bar00] Nicholas Barr. *A.W.H. Phillips: Collected Works in Contemporary Perspective*, chapter The History of the Phillips Machine, pages 89–114. Cambridge University Press, 2000. (Cited on page 8.)
-

- [BBRW03] W.M. Beynon, A.H. Bhalerao, C.P. Roe, and A.T. Ward. A computer-based environment for the study of relational query languages. In *Proc. of the Teaching, Learning and Assessment in Databases Workshop*, pages 104–108, Coventry, UK, July 2003. (079). (Cited on pages xii, 1, 18 and 235.)
- [BBY92] W.M. Beynon, I. Bridge, and Y.P. Yung. Agent-oriented modelling for a vehicle cruise controller. In *Proc. ESDA Conf., ASME PD-Vol. 47-4*, pages 159–165, 1992. (023). (Cited on pages 48 and 52.)
- [BC93] W.M. Beynon and A.J. Cartwright. Agent-oriented modelling for engineering design. In *Proc. CAD '93, New Information Technologies in Science, Education and Business, Yalta*, pages 49–53, May 1993. (030). (Cited on page 89.)
- [BC95] W.M. Beynon and R.I. Cartwright. Empirical modelling principles for cognitive artefacts. In *Proc. IEE Colloquium: Design Systems with Users in Mind: The Role of Cognitive Artefacts, Digest No 95/231, 8/1-8/8*, December 1995. (043). (Cited on page 89.)
- [BCH⁺01] W.M. Beynon, Y-C. Ch'en, H-W. Hseu, S. Maad, S. Rasmeguan, C.P. Roe, J. Rungrattanaubol, S.B. Russ, and A.T. Ward. The computer as instrument. In *Proc. Cognitive Technology '01: Instruments of Mind, LNAI 2117*, University of Warwick, August 2001. Springer-Verlag. (068). (Cited on pages xii, 1 and 14.)
- [BCHW95] Barry Boehm, Bradford Clark, Ellis Horowitz, and Chris Westland. Cost models for future software life cycle processes: COCOMO 2.0. *Annals of Software Engineering*, 1:57–94, 1995. (Cited on page 35.)
- [BCSW99] W.M. Beynon, R.I. Cartwright, P-H. Sun, and A.T. Ward. Interactive Situation Models for Information Systems Development. In *Proc. SCI'99 and ISAS'99, Volume 2, Orlando, USA*, pages 9–16, July 1999. (052). (Cited on pages xii, 1, 5, 14, 22, 48, 50, 287, 292 and 319.)
- [BDMN79] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *SIMULA BEGIN*. Chartwell-Bratt, Old Orchard, Bickley Road, Bromley, Kent BR1 2NE, 1979. (Cited on page 53.)
- [Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, first edition, 1999. (Cited on page 21.)
- [Bey83] W.M. Beynon. A definition of the ARCA notation. Theory of Computation Report CS-RR-054, University of Warwick Department of Computer Science, 1983. (Cited on pages xiv, 229 and 301.)
- [Bey85] W.M. Beynon. Definitive notations for interaction. In *Proceedings of HCI'85*, pages 23–34. Cambridge University Press, 1985. (001). (Cited on page 32.)

- [Bey87a] W.M. Beynon. The LSD notation for communicating systems. Research Report CS-RR-087, University of Warwick Department of Computer Science, 1987. (Cited on pages xv, 47 and 50.)
- [Bey87b] W.M. Beynon. The LSD notation for communicating systems. In *Proc. 3rd British Colloquium for Theoretical Computer Science, Leicester*, 1987. (003). (Cited on page 50.)
- [Bey88] W.M. Beynon. Definitive programming for parallelism. Research Report CS-RR-132, University of Warwick Department of Computer Science, 1988. (Cited on pages 71 and 88.)
- [Bey89a] W.M. Beynon. Definitions as a framework for design. In *Proc. 3rd Eurographics ICAD Workshop, CWI Amsterdam*, 1989. (015). (Cited on pages xiv and 89.)
- [Bey89b] W.M. Beynon. A definitive programming approach to the implementation of CAD software. In *Intelligent CAD Systems II: Implementation Issues*, pages 126–45. Springer-Verlag, 1989. (013). (Cited on pages xiv, 227 and 229.)
- [Bey89c] W.M. Beynon. Evaluating definitive principles for interactive graphics. In *New Advances in Computer Graphics*, pages 291–303. Springer-Verlag, 1989. (011). (Cited on page 88.)
- [Bey90] W.M. Beynon. Parallelism in a definitive programming framework. In *Parallel Computing 89, North Holland: Advances in Parallel Computing Volume 2*, pages 425–430, 1990. (016). (Cited on pages 32, 71, 88 and 162.)
- [Bey94] W.M. Beynon. Agent-oriented modelling and the explanation of behaviour. In *Proc. International W/S “Shape Modeling Parallelism, Interactivity and Applications”, Dept. of Computer Software, TR 94-1-040*, pages 54–63, Univ. Aizu, Japan, September 1994. (037). (Cited on page 89.)
- [Bey97] W.M. Beynon. Empirical Modelling for educational technology. In *Proc. Cognitive Technology '97, University of Aizu, Japan, IEEE*, pages 54–68, 1997. (047). (Cited on pages 50, 89 and 306.)
- [Bey99] W.M. Beynon. Empirical Modelling and the foundations of Artificial Intelligence. *Lecture Notes in Artificial Intelligence*, 1562:322–364, 1999. (050). (Cited on pages 4, 89, 216 and 306.)
- [Bey03] W.M. Beynon. Radical Empiricism, Empirical Modelling and the nature of knowing. In *Proceedings of the WM 2003 Workshop on Knowledge Management and Philosophy*, Luzern, April 3–4 2003. (078). (Cited on page 306.)

- [BF] Dan Bricklin and Bob Frankston. VisiCalc: Information from its creators. <http://www.bricklin.com/visicalc.htm> (accessed April 2004). (Cited on page 10.)
- [BH95] Krishna A. Bharat and Scott E. Hudson. Supporting distributed, concurrent, one-way constraints in user interface applications. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 121–132. ACM Press, 1995. (Cited on page 39.)
- [Bir91] Stuart Bird. An implementation of ARCA in EDEN. Final year undergraduate project report, University of Warwick Department of Computer Science, 1991. (Cited on page 301.)
- [Bis86] Judy Bishop. *Data Abstraction in Programming Languages*. Addison-Wesley, 1986. (Cited on page 23.)
- [Bla02] A. F. Blackwell. First steps in programming: a rationale for attention investment models. In *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pages 2–10, September 2002. (Cited on page 36.)
- [BMR⁺96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A system of patterns*. Wiley, 1996. (Cited on page 39.)
- [BN87] W.M. Beynon and M.T. Norris. Comparison of SDL and LSD. In R. Saracco and P.A.J. Tilanus, editors, *Proc. SDL'87, North Holland*, pages 201–209, 1987. (004). (Cited on pages 50 and 65.)
- [BNOS90] W.M. Beynon, M.T. Norris, R.A. Orr, and M.D. Slade. Definitive specification of concurrent systems. In *Proc. UKIT'90*, pages 52–57. IEE Conference Publications 316, 1990. (017). (Cited on pages 52, 59, 71 and 88.)
- [BNR95] W.M. Beynon, P.E. Ness, and S.B. Russ. Worlds before and beyond words. In *Proc. VF '95*. University of Warwick, 1995. (041). (Cited on page 7.)
- [BNS88] W.M. Beynon, M.T. Norris, and M.D. Slade. Definitions for modelling and simulating concurrent systems. In *Proc. IASTED conference ASM'88*. Acta Press, 1988. (007). (Cited on page 50.)
- [Boo54] George Boole. *An Investigation of the laws of thought on which are founded the mathematical theories of logic and probabilities*. Macmillan, 1854. (Cited on page 26.)
- [BR] W.M. Beynon and S.B. Russ. EM home page. <http://www.dcs.warwick.ac.uk/research/modelling/> (accessed April 2004). (Cited on page 2.)

- [BR92] W.M. Beynon and S.B. Russ. The interpretation of states: a new foundation for computation. In *Proc. PPIG'92, Loughborough*, January 1992. (027). (Cited on page 55.)
- [BR95] W.M. Beynon and S.B. Russ. Empirical Modelling of requirements. Research Report CS-RR-277, University of Warwick Department of Computer Science, 1995. (Cited on page 22.)
- [BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999. (Cited on page 22.)
- [Bro87] Frederick P. Brooks, Jr. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987. (Cited on pages 20, 21, 22 and 27.)
- [Bro01] Chris Brown. An agent-based parsing system in EDEN. Final year undergraduate project report, University of Warwick Department of Computer Science, 2001. (Cited on page xiv.)
- [BRWW01] W.M. Beynon, C.P. Roe, A.T. Ward, and A. Wong. Interactive Situation Models for cognitive aspects of user-artefact interaction. In *Proc. Cognitive Technology '01: Instruments of Mind, LNAI 2117*, pages 356–372, University of Warwick, August 2001. Springer-Verlag. (069). (Cited on pages xii, 1, 10, 43 and 352.)
- [BRY90] W.M. Beynon, S.B. Russ, and Y.P. Yung. Programming as modelling: New concepts and techniques. In *Proc. ISLIP'90. Computing and Information Science Dept, Queen's University, Canada*, 1990. (019). (Cited on pages 65, 89 and 91.)
- [BS99] W.M. Beynon and P-H. Sun. Computer-mediated communication: a distributed Empirical Modelling perspective. In *Proc. of CT'99, San Francisco*, 1999. (053). (Cited on pages 4 and 8.)
- [BSY89] W.M. Beynon, M.D. Slade, and Y.W. Yung. Parallel computation in definitive models. In *CONPAR'88*, pages 359–367. British Computer Society Workshop Series CUP, 1989. (010). (Cited on pages 71 and 88.)
- [BWM⁺00] W.M. Beynon, A.T. Ward, S. Maad, A. Wong, S. Rasmeyuan, and S.B. Russ. The Temposcope: A computer instrument for the idealist timetabler. In Edmund Burke and Wilhelm Erben, editors, *Proc. of the 3rd international conference on the Practice and Theory of Automated Timetabling*, pages 153–175, Fachhochschule Konstanz, University of Applied Sciences, Germany, August 2000. (058). (Cited on pages xii, 1 and 14.)
- [BY90] W.M. Beynon and Y.P. Yung. Definitive interfaces as a visualisation mechanism. In *Proc. Graphics Interface '90*, pages 285–292. Canadian Information Processing Society, 1990. (018). (Cited on pages 89 and 243.)

- [BY92] W.M. Beynon and Y.P. Yung. Agent-oriented modelling for discrete-event systems. In *Proc. IEE Coll. "Discrete-Event Dynamic Systems"*, page Digest 1992/138, 1992. (026). (Cited on page 89.)
- [BY94] W.M. Beynon and Y.P. Yung. A computer-aided script generator for Computer Aided Design. In *Proc. Pacific Graphics '94 / CADDM'94, Vol 2*, pages 369–374, 1994. (039). (Cited on page 89.)
- [BZ89] R. Bubenik and W. Zwaenepoel. Performance of optimistic make. In *Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 39–48. ACM Press, 1989. (Cited on page 39.)
- [Cao02] Charles Cao. A simple complexity — Rubik’s cube. *illuminate*, 0(3), 2002. <http://engrwp.usc.edu/illuminate/article.php?articleID=11>. (Cited on page 227.)
- [Car94] Alan Cartwright. *Application of Definitive Scripts to CACD*. PhD thesis, University of Warwick, July 1994. (Cited on page 18.)
- [Car99] Richard Cartwright. *Geometric Aspects of Empirical Modelling: Issues in Design and Implementation*. PhD thesis, University of Warwick, May 1999. (Cited on pages xiii, xiv, xv, 18, 44, 89, 97, 98, 99, 100, 101, 102, 103, 105, 107, 108, 109, 113, 114, 117, 120, 122, 124, 127, 156, 157, 163, 164, 166, 185, 261, 294, 301, 344 and 345.)
- [Car00] Ben Carter. Sasami project report: A 3D display engine for the EDEN modelling environment. Final year undergraduate project report, University of Warwick Department of Computer Science, 2000. (Cited on pages xv and 224.)
- [Car04] Richard Cartwright. Personal communication. April 2004. (Cited on page 287.)
- [Cas92] Rommert J. Casimir. Real programmers don’t use spreadsheets. *SIGPLAN Not.*, 27(6):10–16, 1992. (Cited on page 34.)
- [Ch’01] Yih-Chang Ch’en. *Empirical Modelling for Participative Business Process Reengineering*. PhD thesis, University of Warwick, December 2001. (Cited on pages 18 and 22.)
- [Cha89] S. Chan. Enhancing the DoNaLD line drawing system. Final year undergraduate project report, University of Warwick Department of Computer Science, 1989. (Cited on page 211.)
- [Che99] Peter Checkland. *Systems Thinking, Systems Practice: Includes a 30-year retrospective*. John Wiley and Sons, 1999. (Cited on pages 4 and 352.)

- [CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with “readers” and “writers”. *Commun. ACM*, 14(10):667–668, 1971. (Cited on page 289.)
- [CJS⁺02] J. Cao, S.A. Jarvis, D.P. Spooner, J.D. Turner, D.J. Kerbyson, and G.R. Nudd. Performance prediction technology for agent-based resource management in grid environments. In *Proc. 11th IEEE Heterogeneous Computing Workshop (HCW02), Marriott Marina, Fort Lauderdale, Florida*, April 2002. (Cited on page 104.)
- [CK93] Paul B. Cragg and Malcolm King. Spreadsheet modelling abuse: An opportunity for OR? *The Journal of the Operational Research Society*, 44(8):743–752, August 1993. (Cited on page 36.)
- [CK03] Martin Campbell-Kelly. *The History of Mathematical Tables: from Sumer to Spreadsheets*, chapter The rise and rise of the spreadsheet, pages 322–347. Oxford University Press, 2003. (Cited on page 33.)
- [CKA96] Martin Campbell-Kelly and William Aspray. *Computer: A History of the Information Machine*. BasicBooks, first edition, 1996. (Cited on page 10.)
- [CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design*. Addison-Wesley, 1988. (Cited on pages xv, 81, 82, 83, 84, 85, 86 and 87.)
- [Coo99] Alan Cooper. *The Inmates are Running the Asylum*. SAMS, A Division of Macmillan Computer Publishing, 201 West 103rd Street, Indianapolis, Indiana 46290, first edition, 1999. (Cited on page 21.)
- [CP03] Steven Carroll and Constantine Polychronopoulos. A framework for incremental extensible compiler construction. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 53–62. ACM Press, 2003. (Cited on page 39.)
- [Cur96] David Curry. *UNIX systems programming for SVR4*. O’Reilly and Associates, 1996. (Cited on page 257.)
- [cvs] Concurrent versions system. <http://www.cvshome.org/> (accessed April 2004). (Cited on page 250.)
- [Dav95] Emma L. Davis. Modelling human interaction. Final year project report, University of Warwick Department of Computer Science, 1995. (Cited on page 76.)
- [DCG⁺89] Peter J. Denning, D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Commun. ACM*, 32(1):9–23, 1989. (Cited on page 352.)

- [DDH72] O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming*, chapter Notes on Structured Programming, pages 1–82. Academic Press Inc. (London) Ltd., 1972. (Cited on page 274.)
- [dHRvE95] Walter A. C. A. J. de Hoon, Luc M. W. J. Rutten, and Marko C. J. D. van Eekelen. Implementing a functional spreadsheet in Clean. *Journal of Functional Programming*, 5(3):383–414, July 1995. (Cited on page 35.)
- [Dij68] Edsger W. Dijkstra. Cooperating Sequential Processes. <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF> (accessed April 2004), 1968. (Cited on pages 84 and 289.)
- [Dij76] Edsger W. Dijkstra. *A discipline of programming*. Prentice-Hall, 1976. (Cited on pages 54, 81, 85, 165 and 352.)
- [Dij01] Edsger W. Dijkstra. The end of computing science? *Commun. ACM*, 44(3):92, 2001. (Cited on page 22.)
- [DW90] Weichang Du and William W. Wadge. The eductive implementation of a three-dimensional spreadsheet. *Softw. Pract. Exper.*, 20(11):1097–1114, November 1990. (Cited on page 35.)
- [exc01] Excel X on-line help, 2001. Microsoft Corporation. (Cited on page 324.)
- [FBY93] Monica Farkas, Meurig Beynon, and Yun Pui Yung. Agent-oriented modelling for a billiards simulation. Research Report CS-RR-260, University of Warwick Department of Computer Science, 1993. (Cited on page 220.)
- [FCR⁺02] Marc Fisher, Mingming Cao, Gregg Rothermel, Curtis R. Cook, and Margaret M. Burnett. Automated test case generation for spreadsheets. In *Proceedings of the 24th international conference on Software engineering*, pages 141–153. ACM Press, 2002. (Cited on page 36.)
- [Fel79] Stuart I. Feldman. Make — a computer program for maintaining computer programs. *Softw. Pract. Exper.*, 9(4):255–265, April 1979. (Cited on page 38.)
- [FH89] C. J. Fleckenstein and D. Hemmendinger. A parallel ‘make’ utility based on Linda’s tuple-space. In *Proceedings of the seventeenth annual ACM conference on Computer science : Computing trends in the 1990’s*, pages 216–220. ACM Press, 1989. (Cited on page 39.)
- [FPB95] Jr. Frederick P. Brooks. *The mythical man-month: essays on software engineering*. Addison Wesley Longman, Inc., 1995. (Cited on pages 21 and 352.)

- [Fri97] Jeffrey E.F. Friedl. *Mastering Regular Expressions*. O'Reilly and Associates, Inc., 1997. (Cited on page 235.)
- [Geh] Dominic Gehring. The MoDD API. <http://www.dcs.warwick.ac.uk/~gehring/modd/> (dated 1998). (Cited on pages 108 and 123.)
- [Geh95] Dominic Gehring. Tables as definitive variables. Final year undergraduate project report, University of Warwick Department of Computer Science, 1995. (Cited on page 301.)
- [GMR95] Ashish Gupta, Inderpal S. Mumick, and Kenneth A. Ross. Adapting materialized views after redefinitions. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 211–222. ACM Press, 1995. (Cited on page 38.)
- [GMT04] Deborah Gage, John McCormick, and Berta Ramona Thayer. Why software quality matters. *Baseline*, pages 34–59, March 2004. (Cited on page 20.)
- [Gol] Jody Goldberg. The Gnumeric GNOME desktop environment spreadsheet. <http://www.gnome.org/projects/gnumeric/> (accessed August 2004). (Cited on page 34.)
- [Goo01] David C. Gooding. Experiment as an instrument of innovation: Experience and embodied thought. In M. Beynon, C.L. Nehaniv, and K. Dautenhahan, editors, *Cognitive Technology: Instruments of Mind*, pages 130–140. Springer-Verlag, 2001. (Cited on page 6.)
- [GP96] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A ‘cognitive dimensions’ framework. *Journal of Visual Languages and Computing*, 7(2):131–174, June 1996. (Cited on page 36.)
- [GS93] A. Gibbons and P. Spirakis, editors. *Lectures on Parallel Computation*. Cambridge International Series on Parallel Computation. Cambridge University Press, 1993. (Cited on page 108.)
- [GYC⁺96] D.K. Gehring, Y.P. Yung, R.C. Cartwright, W.M. Beynon, and A.J. Cartwright. Higher-order constructs for interactive graphics in a definitive programming framework. In *Proc. 14th Eurographics UK Conference*, pages 179–192, 1996. (044). (Cited on pages 98 and 325.)
- [Hal01] Alon Y. Halevy. Answering queries using views: a survey. *The VLDB Journal*, 10(4):270–294, 2001. (Cited on page 38.)
- [Han02a] Keith Hanna. Interactive visual functional programming. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 145–156. ACM Press, 2002. (Cited on page 35.)

- [Han02b] Per Brinch Hansen. *The origin of concurrent programming: from semaphores to remote procedure calls*, chapter The invention of concurrent programming, pages 3–61. Springer-Verlag New York, Inc., 2002. (Cited on pages 24 and 68.)
- [Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987. (Cited on page 16.)
- [Har88] David Harel. On visual formalisms. *Commun. ACM*, 31(5):514–530, 1988. (Cited on page 309.)
- [Har02] Antony Harfield. Agent-oriented parsing with Empirical Modelling. Final year undergraduate project report, University of Warwick Department of Computer Science, 2002. (Cited on page xiv.)
- [Haz] Philip Hazel. PCRE — Perl Compatible Regular Expressions library. <http://www.pcre.org/> (accessed February 2004). (Cited on page 235.)
- [her00] The American Heritage dictionary of the English language, 2000. <http://dictionary.reference.com/> (accessed April 2004). (Cited on page 3.)
- [Her02] Timothy Heron. Programming with dependency. Master’s thesis, University of Warwick, September 2002. (Cited on pages 39, 89 and 261.)
- [HNC65] Frank Harary, Robert Z. Norman, and Dorwin Cartwright. *Structural Models: An Introduction to the Theory of Directed Graphs*. John Wiley and Sons, 1965. (Cited on pages 106, 179 and 180.)
- [Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, 1978. (Cited on page 84.)
- [HP03] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2003. (Cited on page 220.)
- [Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989. (Cited on pages 30 and 40.)
- [Hud94] Scott E. Hudson. User interface specification using an enhanced spreadsheet model. *ACM Trans. Graph.*, 13(3):209–239, 1994. (Cited on pages 36 and 37.)
- [IEE] IEEE. IEEE Xplore. <http://ieeexplore.ieee.org/> (accessed August 2004). (Cited on page 34.)
- [IL85] Kaoru Ishikawa and David J. Lu. *What is total quality control? : the Japanese way*. Prentice Hall, 1985. (Cited on page 4.)

- [ISL95] Tomás Isakowitz, Shimon Schocken, and Henry C. Lucas, Jr. Toward a logical/physical theory of spreadsheet modeling. *ACM Trans. Inf. Syst.*, 13(1):1–37, 1995. (Cited on page 36.)
- [itr] International Technology Roadmap for Semiconductors 2003 edition. <http://public.itrs.net/Files/2003ITRS/Home2003.htm> (accessed April 2004). (Cited on page 220.)
- [Jam12] William James. *Essays in Radical Empiricism*. Longmans, Green and Co., 1912. (Cited on pages 3 and 306.)
- [JBB03] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. A user-centred approach to functions in Excel. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 165–176. ACM Press, 2003. (Cited on page 36.)
- [JC92] Ivar Jacobson and Magnus Christensen. *Object-oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992. (Cited on page 21.)
- [Kay84] Alan Kay. Computer software. *Scientific American*, 251(3):52–59, September 1984. (Cited on page 34.)
- [KB00] I. Kalas and A. Blaho. Imagine... new generation of Logo: Programmable pictures. In *Proc. of WCC2000, Beijing*, pages 427–430, 2000. (Cited on page 242.)
- [Ken78] William Kent. *Data and Reality: basic assumptions in data processing reconsidered*. North-Holland Publishing Company, 1978. (Cited on page 11.)
- [Kin04] Karl King. Timetabling with Empirical Modelling principles and tools. Final year undergraduate project report, University of Warwick Department of Computer Science, 2004. (Cited on page 249.)
- [Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, second edition, 1973. (Cited on pages 44, 106 and 108.)
- [KP74] Brian W. Kernighan and P.J. Plauger. *The elements of programming style*. McGraw-Hill, 1974. (Cited on page 352.)
- [KP84] Brian W. Kernighan and Rob Pike. *The UNIX programming environment*. Prentice Hall, 1984. (Cited on pages 196 and 197.)
- [KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988. (Cited on page 39.)

-
- [KPP⁺02] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28(8):721–734, 2002. (Cited on page 352.)
- [Lev95] Nancy G. Leveson. *Safeware: System safety and computers*. Addison-Wesley, September 1995. (Cited on pages 26 and 349.)
- [Lew90] C. Lewis. *Visual Programming Environments: Paradigms and Systems*, chapter NoPumpG: Creating interactive graphics with spreadsheet machinery, pages 526–546. IEEE Computer Society Press, 1990. (Cited on page 37.)
- [LM02] Björn Lisper and Johan Malmström. Haxcel: A spreadsheet interface to Haskell. In *Proceedings of the 14th International Workshop on the Implementation of Functional Languages*, pages 206–222, September 2002. (Cited on page 35.)
- [LT77] Henry F. Ledgard and Robert W. Taylor. Two views of data abstraction. *Commun. ACM*, 20(6):382–384, 1977. (Cited on page 23.)
- [Maa02] Soha Maad. *An Empirical Modelling Approach to Software System Development in Finance: Applications and Prospects*. PhD thesis, University of Warwick, March 2002. (Cited on page 18.)
- [McC] G. Frank McCormick. When reach exceeds grasp. (Cited on pages 26 and 349.)
- [Mez87] Samia Meziani. Denota - an interpreter for definitive notations. Master’s thesis, University of Warwick, December 1987. (Cited on pages 45, 301, 303 and 347.)
- [Mil93] Robin Milner. Elements of interaction: Turing award lecture. *Commun. ACM*, 36(1):78–89, 1993. (Cited on page 28.)
- [Mil02] Joaquin Miller. What UML should be: introduction. *Commun. ACM*, 45(11):67–69, 2002. (Cited on page 22.)
- [Moo65] Gordon E. Moore. Cramming more circuits onto integrated circuits. *Electronics*, 38(8), April 1965. (Cited on page 220.)
- [Mor] MoreSteam.com. The Cause and Effect “Fishbone” diagram. <http://www.moresteam.com/toolbox/t406.cfm> (dated 2000). (Cited on pages 4 and 5.)
- [mot] Digital Performer: Persistent unlimited multiple undo / redo with branching. MOTU, Inc <http://www.motu.com/english/software/dp/dp3/undo.html> (dated 2001). (Cited on page 250.)
-

- [Nar93] Bonnie A. Nardi. *A small matter of Programming: Perspectives on End User Computing*. MIT Press, 1993. (Cited on pages 35 and 36.)
- [Nes97] Paul Edward Ness. *Creative Software Development: An Empirical Modelling Framework*. PhD thesis, University of Warwick, October 1997. (Cited on page 18.)
- [Neu] Peter G. Neumann. RISKS-LIST. <http://catless.ncl.ac.uk/Risks> (accessed March 2004). (Cited on page 12.)
- [Neu95] Peter G. Neumann. *Computer Related Risks*. The ACM Press, 1995. (Cited on page 20.)
- [NM90] Bonnie A. Nardi and James R. Miller. An ethnographic study of distributed problem solving in spreadsheet development. In *Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, pages 197–208. ACM Press, 1990. (Cited on page 35.)
- [Ope] OpenOffice.org. OpenOffice productivity suite. <http://www.openoffice.org/> (accessed August 2004). (Cited on page 34.)
- [Oun04] Asma Ounnas. Development of the EDDI parser documentation of eddi.eden. Final year undergraduate project report, University of Warwick Department of Computer Science, 2004. (Cited on page 233.)
- [Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994. (Cited on pages xv and 214.)
- [Par91] J. Parsons. Enhancement of the DoNaLD translator. Final year undergraduate project report, University of Warwick Department of Computer Science, 1991. (Cited on page 211.)
- [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. (Cited on page 98.)
- [Per] Perforce Software, Inc. Jam. <http://www.perforce.com/jam/jam.html> (accessed August 2004). (Cited on page 39.)
- [Per01] Michael L. Perry. Automate dependency tracking, 2001. http://www.javaworld.com/javaworld/jw-08-2001/jw-0817-automatic_p.html (dated 17 August 2001). (Cited on page 39.)
- [PH98] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware / Software Interface*. Morgan Kaufmann, second edition, 1998. (Cited on page 156.)
- [Phi00] A.W. Phillips. *A.W.H. Phillips: Collected Works in Contemporary Perspective*, chapter Mechanical Models in Economic Dynamics, pages 68–88. Cambridge University Press, 2000. (Cited on pages xiii and 8.)

- [Pie86] Kurt W. Piersol. Object-oriented spreadsheets: the analytic spreadsheet package. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 385–390. ACM Press, 1986. (Cited on page 33.)
- [PKNA95] E. Papaefstathiou, D.J. Kerbyson, G.R. Nudd, and T.J. Atherton. An overview of the CHIP³S performance prediction toolset for parallel systems. In *Proc. of 8th IEEE International Conference on Parallel and Distributed Computing Systems, Orlando, Florida, USA*, September 1995. (Cited on page 104.)
- [Puc87] Tom Puckett. Implementation of an APL-based spreadsheet manager. In *Proceedings of the international conference on APL*, pages 163–172. ACM Press, 1987. (Cited on page 33.)
- [Ram] Chet Ramey. GNU Readline library. <http://www.gnu.org/directory/readline.html> (accessed April 2004). (Cited on page 249.)
- [Ras01] Suwanna Rasmequan. *An Approach to Computer-based Knowledge Representation for the Business Environment using Empirical Modelling*. PhD thesis, University of Warwick, November 2001. (Cited on page 18.)
- [Rau96] Martin Frank Rausch. The agent repository — supporting collaborative contextualized learning with a medium for indirect communication. Master’s thesis, University of Colorado, Department of Computer Science, 1996. (Cited on page 37.)
- [Ray] Eric Steven Raymond. The cathedral and the bazaar. <http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/index.html> (dated September 2000). (Cited on pages 21 and 223.)
- [Rep93] A. Repenning. *Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments*. PhD thesis, University of Colorado at Boulder, Department of Computer Science, December 1993. (Available as Technical Report CU-CS-693-93). (Cited on page 37.)
- [Roe03] Chris Roe. *Computers for Learning: An Empirical Modelling Perspective*. PhD thesis, University of Warwick, November 2003. (Cited on pages 7, 18, 218 and 249.)
- [Rol82] L.T.C. Rolt. *Red for Danger*. Pan Books, fourth edition, 1982. (Cited on pages 4 and 216.)
- [RR93] G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 502–510. ACM Press, 1993. (Cited on page 40.)

- [Run02] Jaratsri Rungrattanaubol. *A Treatise on Modelling with Definitive Scripts*. PhD thesis, University of Warwick, April 2002. (Cited on pages 19, 46, 56, 78, 88 and 89.)
- [San] Georg Sander. VCG overview. <http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html> (dated September 1995). (Cited on page 180.)
- [San96] Georg Sander. *Visualisierungstechniken für den Compilerbau*. PhD thesis, Universität des Saarlandes, Technische Fakultät, 66125, 1996. (Cited on page 180.)
- [Sed90] Robert Sedgewick. *Algorithms in C*. Addison-Wesley, 1990. (Cited on page 186.)
- [Sha03] William Shakespeare. *The Tragedy of Hamlet*, chapter Scene I. A room in the castle. Folio edition, 1603. (Cited on page 152.)
- [She81] B. A. Sheil. The psychological study of programming. *ACM Comput. Surv.*, 13(1):101–120, 1981. (Cited on page 352.)
- [ŠilcRU01] Jurij Šilc, Borut Robič, and Theo Ungerer. *Progress in computer research*, chapter Asynchrony in parallel computing: from dataflow to multithreading, pages 1–33. Nova Science Publishers, Inc., 2001. (Cited on pages 30 and 40.)
- [Sla90] Mike Slade. Definitive parallel programming. Master’s thesis, University of Warwick, April 1990. (Cited on pages xiii, xiv, 43, 46, 47, 52, 53, 55, 56, 58, 59, 60, 62, 63, 64, 67, 68, 69, 70, 78, 80, 89, 310 and 344.)
- [Slo] N. J. A. Sloane. The on-line encyclopedia of integer sequences. <http://www.research.att.com/~njas/sequences/> (accessed March 2004). (Cited on page 184.)
- [sta] StarOffice office suite. Sun Microsystems, Inc. <http://www.sun.com/software/star/staroffice/> (accessed August 2004). (Cited on page 34.)
- [Sun99] Patrick Pi-Hwa Sun. *Distributed Empirical Modelling and its Application to Software System Development*. PhD thesis, University of Warwick, July 1999. (Cited on pages xiv, xv, 4, 12, 18, 22, 62, 89, 216 and 326.)
- [Tan99] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice Hall, fourth edition, 1999. (Cited on page 31.)
- [TBH82] Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins. Data-driven and demand-driven computer architecture. *ACM Comput. Surv.*, 14(1):93–143, 1982. (Cited on page 60.)

- [TBT00] Duane Truex, Richard Baskerville, and Julie Travis. Amethodical systems development: the deferred meaning of systems development methods. *Accounting, Management and Information Technology*, 10:53–79, 2000. (Cited on page 352.)
- [Tic85] Walter F. Tichy. RCS — a system for version control. *Softw. Pract. Exper.*, 15(7):637–654, 1985. (Cited on page 250.)
- [Til43] E.M.W. Tillyard. *The Elizabethan world picture*. Chatto and Windus, London, 1943. (Cited on page 6.)
- [Tod76] S.J.P. Todd. The Peterlee relational test vehicle — a system overview. *IBM Systems Journal*, 15(4):285–308, 1976. (Cited on page 229.)
- [Tru96] S.V. Truong. Interfacing EDEN with ORACLE. Final year undergraduate project report, University of Warwick Department of Computer Science, 1996. (Cited on pages xiv and 229.)
- [usb] Universal Serial Bus home. USB Implementers Forum, Inc. <http://www.usb.org> (accessed April 2004). (Cited on page 250.)
- [Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990. (Cited on page 70.)
- [VHM⁺01] Bradley T. Vander Zanden, Richard Halterman, Brad A. Myers, Rich McDaniel, Rob Miller, Pedro Szekely, Dario A. Giuse, and David Kosbie. Lessons learned about one-way, dataflow constraints in the Garnet and Amulet graphical toolkits. *ACM Trans. Program. Lang. Syst.*, 23(6):776–796, 2001. (Cited on page 39.)
- [VL02] Lionel Villard and Nabil Layaïda. An incremental XSLT transformation processor for XML document manipulation. In *Proceedings of the eleventh international conference on World Wide Web*, pages 474–485. ACM Press, 2002. (Cited on page 40.)
- [VMGS94] Brad Vander Zanden, Brad A. Myers, Dario A. Giuse, and Pedro Szekely. Integrating pointer variables into one-way constraint models. *ACM Trans. Comput.-Hum. Interact.*, 1(2):161–213, 1994. (Cited on pages 39 and 41.)
- [VP95] Lawrence G. Votta and Adam Porter. Experimental software engineering: a report on the state of the art. In *Proceedings of the 17th international conference on Software engineering*, pages 277–279. ACM Press, 1995. (Cited on page 351.)
- [WA85] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language*, volume 22 of *A.P.I.C. Studies in Data Processing*. Academic Press Inc. (London) Ltd., 1985. (Cited on page 29.)

- [Wara] A.T. Ward. The EDEN change.log file. <http://www.dcs.warwick.ac.uk/research/modelling/tools/eden-change.log.html> (accessed April 2004). (Cited on page 223.)
- [Warb] A.T. Ward. EDEN SourceForge.net project. <http://sourceforge.net/projects/eden/> (dated December 2000). (Cited on page 1.)
- [Warc] A.T. Ward. EM Tools home page. <http://www.dcs.warwick.ac.uk/research/modelling/tools/> (accessed April 2004). (Cited on page 1.)
- [War98] A.T. Ward. EWE - the Empirical Wool Environment. Taught MSc project, University of Warwick Department of Computer Science, 1998. <http://www.dcs.warwick.ac.uk/~ashley/EWE/> (dated November 1998). (Cited on pages 18 and 123.)
- [Wat] Gray Watson. Dmalloc - a debug malloc library. <http://dmalloc.com/> (accessed April 2004). (Cited on page 228.)
- [Weg97] Peter Wegner. Why interaction is more powerful than algorithms. *Commun. ACM*, 40(5):80–91, 1997. (Cited on page 29.)
- [Wil69] John Wild. *The Radical Empiricism of William James*. Greenwood Press, Westport, Connecticut, 1969. (Cited on pages 3, 4, 23 and 306.)
- [Wil96] Robin J. Wilson. *Introduction to Graph Theory*. Longman, fourth edition, 1996. (Cited on pages 184 and 282.)
- [Wil02] Maurice V. Wilkes. Moore's law and the future. <http://www.cl.cam.ac.uk/users/mvw1/lect-Moores-law-and-the-future.pdf> (accessed April 2004), October 2002. (Cited on page 220.)
- [Wir76] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall, 1976. (Cited on pages 36 and 235.)
- [WL90] Nicholas Wilde and Clayton Lewis. Spreadsheet-based interactive graphics: from prototype to tool. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 153–160. ACM Press, 1990. (Cited on page 37.)
- [Won03] Allan Kai Tung Wong. *Before and Beyond Systems: an Empirical Modelling approach*. PhD thesis, University of Warwick, January 2003. (Cited on pages 18, 89, 182 and 247.)
- [WRB] A.T. Ward, C.P. Roe, and W.M. Beynon. empublic project archive. <http://empublic.dcs.warwick.ac.uk/projects/> (accessed April 2004). (Cited on pages xv, 1, 18, 20, 37, 108, 201 and 218.)

- [WS97] Laura Wingerd and Christopher Seiwald. Constructing a large product with Jam. In *Proceedings of the SCM-7 Workshop on System Configuration Management*, pages 36–48. Springer-Verlag, 1997. (Cited on page 39.)
- [WW90] Robin J. Wilson and John J. Watkins. *Graphs: An Introductory Approach*. John Wiley and Sons, 1990. (Cited on pages 184 and 185.)
- [YC94] Alan G. Yoder and David L. Cohn. Real spreadsheets for real programmers. In *Proceedings of the 1994 IEEE International Conference on Computer Languages*, pages 20–30. IEEE Press, May 1994. (Cited on page 35.)
- [YS91] Daniel M. Yellin and Robert E. Strom. INC: a language for incremental computations. *ACM Trans. Program. Lang. Syst.*, 13(2):211–236, 1991. (Cited on page 40.)
- [Yun87] Edward Yung. EDEN - Evaluator of DEfinitive Notations. Final year undergraduate project report, University of Warwick Department of Computer Science, 1987. (Cited on pages 196, 199, 200, 201, 219 and 259.)
- [Yun89] Edward Y.W. Yung. The EDEN handbook, 1989. (Cited on pages xiv, 32 and 194.)
- [Yun90] Edward Yun Wai Yung. EDEN: An Engine for Definitive Notations. Master's thesis, University of Warwick, September 1990. (Cited on pages xiii, xiv, xv, 44, 102, 184, 194, 196, 199, 200, 210, 211, 243, 259, 260, 275, 284, 286, 287, 288 and 344.)
- [Yun93] Simon Yun Pui Yung. *Definitive Programming - a Paradigm for Exploratory Programming*. PhD thesis, University of Warwick, January 1993. (Cited on pages xiii, xv, 18, 48, 50, 51, 52, 62, 64, 66, 89, 196, 210, 212, 260 and 275.)
- [Yun96] Simon Yun Pui Yung. Agent-oriented modelling for interactive systems. Post-doctoral EPSRC project report, University of Warwick Department of Computer Science, July 1996. (Cited on pages 45, 62, 76, 77, 89, 196, 214, 274 and 275.)

Index

(Note: this index has not been constructed to a professional standard and should not be relied on for completeness — for example, the list of page numbers given after each entry may omit important references in some cases. It is hoped, however, that the index may help to find some information of interest.)

Symbols

!Donald, 122–141, 149, 152, 155–163
 DAMscript, 124, 125, 127–131,
 133, 135, 135_{fn}, 140, 141, 143,
 146, 147, 149, 151, 153, 156,
 159, 163, 164, 169–171, 173,
 174
 definition of, 129
 proposal for ‘explicit context’
 redesign, 147
 reason for name, 124_{fn}
 translation from DoNaLD, 127
 graphical actions, 123, 124,
 130–132, 135, 137, 139, 140,
 144–146, 148, 149, 151, 153,
 155, 159, 161
 graphical interface, 125
 literal values, 126, 140, 145
 primary sources, 98, 124
 stages of translation, 126, 127
 symbol table, 97, 116_{fn}, 124, 130,
 131, 135_{fn}, 148, 149, 152, 155,
 162, 164
 !Donald2, 140–161, 164_{fn}, 166, 170,
 174
 extensions over !Donald, 140
 graphical interface, 141, 145
 operators, 141

1-agent
 model, 8
 modelling, 52–54, 97, 209
3dexoRoe2001, 227

A

A action store, see under ADM
 AADM, 41, 46, 78
 Abstract Definitive Machine, see
 ADM
 abstraction, 23, 30, 162
 shown graphically in DMT, 182
 accident investigations, 4, 7, 20, 216
 Acorn, 110, 156–158, 162
 action
 translating to definition, see
 under definition
 actions
 sequences of, 24
 minor errors cause major
 problems, 25
 triggered, in EDEN, see under
 EDEN
 acyclic graph, directed, see script
 graph
 ADM, 46–90, 194, 195
 ‘Abstract’ epithet, 63
 Abstract Definitive Modelling

- framework
 - (Rungrattanaubol), 46, 78, 89
- action store A , 55, 61
- actions, 54
 - cf LSD privileges, 54
- animating LSD account, see under LSD
- as machine, 46
- command lists, 69, 75, 78
 - cf UNITY conditional assignments, 85
 - divisible, 74, 90
 - or sets?, 70–81, 121
- commands, 53
- context-dependent error, 67
- definition store D , 53
- definitions, 52
 - cf UNITY transparent variables, 87
- distinguishing from LSD, 59, 89
- embodiment, 62
- entities, 53, 55, 59, 196
 - cf OO, 53
 - instantiation, 55
- entity
 - instantiation, 53, 55, 62, 70, 74
- evaluation, 71, 76
 - within redefinition, 52, 53, 56
- external observer perspective, 58
- global clock, 59
- guards, 54, 58
 - evaluation, 61, 69, 71
 - used to simulate control flow, 195
- invalid transition, 67–69, see also under invalid transition
- machine cycle, 109
- modes of execution, 80
- non-determinism, 78, 80, 86
- output, 53, 62
- papers and theses, 88
- parallel execution, 67, 68, 71
- program store P , 53, 55
- redefinitions in, 52
- run set, 68
- scripts, 54, 55
 - cf LSD accounts, 58–59
- state
 - global, 58
 - initial state S , 72, 119
 - intermediate states S^* , 72, 77, 78
 - resultant state S' , 72
 - transition, see also under invalid transition
- super-agent, 80
- transition, 52, 69–72, 77
 - major/minor, 72, 119
- adm, 62, 76, 89, 90
- adm2, 62, 76, 89, 90
- adm3, 62, 89, 90
- ADT, 24
- agency, see also under dependency
 - “centres of state change”, 65
 - attributing in general, 67
 - automated, 54, 68
 - classified using LSD, 50
 - definition of, 16
 - distinction with dependency, 42, 269
 - in AADM, 78
 - in EDEN and spreadsheet, 16
 - lack of prominence in DAM machine, 97
 - most primitive understanding, 6
 - of redefinition, 53
 - super-agent, see under ADM
 - three views of understanding, see under LSD, agent
 - used to construct EDEN, 41
- agent, see also under EDEN, AOP
 - definition of, 5
- Agent Repository, 37
- Agent-Oriented Parser, see AOP
- agentparserBrown2001*, 235
- agentparserHarfield2003*, 235–238
- Agentsheets, 37
- am, 62, 69, 72, 76, 89, 90, 93–96, 108, 194

- cat flap model, 91
 - evaluation/storage strategy, 63
 - `print`, 62
 - Amulet, 39
 - analytical reduction
 - limits on, 4, 23
 - animism, 6_{fn}
 - AOP, see under EDEN
 - Apple
 - Mac OS X, 1, 221, 227_{fn}
 - Macintosh, xi, 220, 324
 - ARCA, 211, 212, 229, 256, 301
 - arcaBird1991*, 212, 256
 - arcaWard2002*, 212
 - ARM processor, 110
 - register, 110, 112, 120–122, 165
 - artefacts, 6, 11
 - assembler, 87, 97, 111, 112, 120, 124, 127, 131, 149, 155, 162
 - Authentic ADM, see AADM
- B** _____
- backroomWard2002*, 1
 - Backus, John, see under software
 - badger, self-referential, 374
 - based on ISBL, 229
 - BASIC, 28_{fn}, 111, 124, 135_{fn}, 149–155, 162, 164_{fn}, 166_{fn}, 173
 - bc, 197
 - billiardsCarter1999*, 227
 - billiardsYung1996*, 220
 - block redefinition, see under redefinition
 - Bool, George, see Boolean algebra
 - Boolean algebra, 26, 349
 - British Telecom Research Laboratories, 47
 - Brooks, Frederick, see under software
 - BSP, 70
- C** _____
- carparkingsimMcHale2003*, 227, 251, 252
 - catflapWard1997*, 1
 - cause and effect, 4, 178, 318
 - Cayley, Arthur, see ARCA
 - Chain of Being, 6
 - change propagation, see propagation
 - of change
 - circuits, 26–32, 41, 349
 - Clayton Tunnel, 216
 - claytontunnelSun1999*, 216–218
 - Cognitive Dimensions of notations, 36, 351
 - complexity, 4, see under software
 - computer
 - as instrument, 14
 - von Neumann, 29, 40, 41, 63, 84, 147, 216
 - von Neumann bottleneck, 63
 - concurrency
 - abstractions for, 24
 - described by LSD, 59
 - not exploited by DAM machine, 119
 - concurrent
 - agent protocols, 289, 292
 - agent roles, 288, 293, 304
 - constraint satisfaction, 39
 - definition maintenance, 184, 280–294, 304–327
 - ‘curtains’, 293
 - ‘rays’, 294
 - disjoint processes, 68
 - execution, 77
 - synchronisation, 32, 280, 287–294, 310, 321
 - mutual exclusion problem, 289
 - readers/writers problem, 289
 - UNITY computational model, 81
 - constraints
 - dataflow, 39
 - distinction with dependency, 178
 - one-way, 39, 209
 - satisfaction algorithm, 39
 - construal, 6
 - cruisecontrolBridge1991*, 214, 215
 - cubesymWong2001*, 227
 - CVS, 250

D

-
- D* definition store, see under ADM
 - D* dependency argument mapping, see under DAM machine
 - dag, 179
 - DAM
 - BRA
 - no advantage over EDEN, 265
 - DAM machine, 97–177, 194, 195, 201, 259, 262, 287, 306
 - addtoq, 117, 119–122, 124, 127, 130, 137, 155, 195
 - applications, 98
 - arrays, 122, 164, 165, see also lists
 - block of redefinitions *K*, 103, 105, 107, 117, 119, 121, 127
 - Block Redefinition Algorithm, see BRA
 - BRA, 66, 103–109, 112, 119, 127, 161, 195, see also redefinition, block
 - efficiency, 107, 108
 - Knuth Counters, 107, 114, 116, 117, 166, 169
 - Knuth’s algorithm, 106, 107
 - purpose, 105
 - update, 103–109, 113, 117, 119, 121, 122, 127, 137, 166, 170
 - cyclic dependency, 103, 106
 - data structure, 113–117
 - Definitive Store, 114, 116, 122, 123, 131
 - definitive store, 112
 - dependency argument mapping *D*, 102, 114
 - dependency function mapping *F*, 102, 114
 - Function Pointer, 114, 117, 121
 - Function Store, 114, 116
 - lists, 101_{fn}, 165, see also arrays
 - literal values, 117, 121, 129, 135, 147, 148, 152, 154, 155, 165, 174
 - monolithic definition-agent, 284, 321
 - operators, 112
 - add, 113, 116
 - contrast with ‘functions’, 112_{fn}
 - distance, 124
 - fif_mult, 146
 - fn, 149, 152
 - gfn, 149, 151
 - if, 147, 148, 155, 164
 - in !Donald2, 141
 - line, 137
 - lookup, 164, 165
 - side effect, 122–124, 127, 129, 137, 139, 155, 195
 - platform, 110
 - primary and secondary sources, 98
 - protected update, 103, 106
 - re-entrancy, 122, 195
 - removing definitions, 103, 148
 - set of functions *F*, 100–102, 104, 112
 - Source Pointer, 114, 116, 121, 166
 - Sources List Pointer, 114, 117
 - Sources Store, 114, 116, 117, 165
 - suitable *K*, 103
 - symbol table, lack of, 116
 - Target Pointer, 114, 116, 117
 - Targets List Pointer, 116, 117
 - Targets Store, 116, 117, 165
 - redundant, 321
 - undefined values, 103
 - Value, 100, 114, 116, 117, 121
 - data flow
 - constraints, see under constraints
 - hardware, 40
 - languages, 30, 40
 - data-driven, see under evaluation
 - strategies
 - database, see also EDDI
 - atomic updates cf DAM machine
 - update, 121
 - dependency in, 37
 - formalisation, 11
 - materialized view, 37

- Oracle, connected to EDDI, 231
- definition
 - agents, 281
 - action perceived as indivisible, 288
 - compare multicomputer node, 285
 - constraints on action, 270, 305, 310
 - definition of, 270
 - monolith, 284
 - script graph node, 282
 - as data+program, 285
 - as formalised in LLDN, 99
 - as function call, 60
 - as guarantee, 288, 292, 348
 - definition of, 16, 30, 42
 - distinction with action, 269
 - higher-order, see HOD
 - in ADM, see under ADM
 - referentially transparent, 233
 - translating to action, 265, 282
 - non-1-1 form, 283
 - one-to-one form, 283
 - when to evaluate, see evaluation strategies
- definitive
 - notations
 - ‘pure’, definition of, 209
 - definition of, 18
 - domain-specific, 209
 - implementing in EDEN, 210, 231–242
 - implications of tri-box framework, 323
 - program, see under definitive script
 - script
 - as formalised in LLDN, 99
 - cf spreadsheet, 26
 - definition of, 16
 - drawing, 178
 - drawn as rectangle, 100
 - extensibility, 27
 - implied sequential meaning, 100
 - interpreted as a relation, 100
 - of Batcher bitonic merge sort, 108
 - symbolic, see under definitive state
 - to find min and max values, 107
 - vs definitive program, 52^m
 - scripts
 - composing, 30
 - state, 52, see also state considered timeless, 53
 - symbolic, 305
 - systems
 - difficulty of comparing, 161
- Definitive Assembly Maintainer machine, see DAM machine
- Definitive Modelling Framework model, see DMF
- Definitive Notation for Line Drawing, see DoNaLD
- Definitive State Transition model, see DST
- definitivedmWard2001*, 1
- demand-driven, see under evaluation strategies
- dependable, see under software
- dependants
 - ‘targets’ preferred, 184
- dependees
 - ‘sources’ preferred, 184
- dependencies
 - ‘sources’ preferred, 102
- dependency
 - ‘instantaneous’ abstraction, 30
 - and foundations of programming, 352
 - as agency, 42, 282, 350
 - as an abstraction, 19, 20, 23, see abstraction
 - as relationship, 23, 26, 27, 97
 - conflates program and data, 19
 - cyclic, 33, 263, see also under DAM machine

- definition of, 5, 6, 42
 - distinction with agency, 42
 - examples, 42
 - functional
 - as formalised in LLDN, 100
 - graph, 33, 40, see also script graph
 - in EDEN and spreadsheet, 16
 - in hardware, 64, 256
 - maintenance, see evaluation strategies
 - many-to-many, 241
 - properties of abstraction, 20
 - subjective perception, 288, 305, 319, 326–327
 - used at a lower level, 41
 - Dependency Maintainer Model, see DMM
 - Dependency Modelling Tool, see DMT
 - dependents
 - ‘targets preferred’, 184
 - ‘targets’ preferred, 102
 - Descartes, 4
 - diagrammatic form
 - circuits, 27
 - definitive script, see script graph
 - software, see under software
 - digital watch interface ISM, 14
 - directed acyclic graph, see script graph
 - distributed
 - constraint satisfaction, 39
 - DMF, 89
 - DMM, 97–110, 112–114, 119, 121, 123, 162, 306
 - assumptions, 101, 108
 - literal values, 101, 106
 - machine cycle, 109
 - primary source, 98
 - reason for abbreviation, 99
 - DMT, 182
 - DoNaLD, 89, 98, 123–127, 131–133, 135, 140, 141, 155, 156, 163, 166, 194, 210–212, 214, 216, 219, 224, 225, 227–229, 231, 233, 237, 240, 247, 253, 255, 260
 - definition of, 127, 209
 - lists in Eden translation, 294
 - object layering, 139
 - strongly typed, 228
 - undue prominence in `tkeden`, 229
 - DST, 89
- ## E
-
- EDDI, 229–232
 - connected to Oracle, 231
 - example of, 18
 - SQL, 235
 - eddip*, see under EDDI
 - eddipTruong1996*, 231, 235
 - eddipWard2000*, 231
 - eddirTruong1996*, 231
 - EDEN, 16, 194–276
 - “in itself”, 208, 223, 229, 241, 247, 250
 - actions, 195, 199, 200, 203–204, 208, 210, 212, 225, 260, 263, 274, 275
 - ‘equivalence’ with definitions, 267
 - alternative syntax, 200, 203
 - cf ADM guards, 195
 - changing the script graph, 275
 - definition of, 200
 - definitions translated into, 267
 - definitions translated to, see under definition
 - distinction with definitions, 269
 - encoding, 270
 - execution not interleaved, 195
 - implicit and explicit attempted distinctions, 260–262
 - interleaving, 213
 - place in stack of virtual machines, 271
 - to implement HOD, 276
 - triggered by causal change, 200
 - ADM to EDEN translators, 62

- AOP, 231–239
 - agent, 236
 - driven by dependency, 240, 318
 - EDDI, see under EDDI
 - LOGO parser, 235
 - palindrome parser, 235, 239
 - PL/0 parser, 235
 - re-entrant parsers, 240
 - SQL parser, 235
- architecture, 216
 - key issues raised, 213
- as an OS, 245
- audience, 221, 228, 247
- `autocalc`, 66, 105, 161, 199, 262, 265, 271
- block redefinition, 105
- clocking procedure, 255
- code growth, 218
- comparison with spreadsheet, 16
- differentiated from Eden, 194, 195
- disadvantages of extended with procedural code, 227
- DoNaLD, see DoNaLD
- `dtkeden`, 41, 216–219, 251, 253, 326
 - God’s eye view, 216
- EDEN/X, see EX
- `eden0`, 299
- `edens1`, 298
- effect of increased processor performance, 220
- EX, 212, 214, 219
- `execute` keyword, 214, 227, 229, 240, 243, 295_m, 299
- Formula Variables, see FVs
- functions, 204
- FVs, 200, 203, 207, 208
 - definition of, 199
 - distinction with RWVs, 199
 - examples of, 207
 - substituted for RWVs, 207, 208
- history, 196–199, 209, 211–223
- implementation, 41, 105
- Interactive Process Translator, 257
 - `is` keyword, 194, 199, 224, 269
 - lacking features of the ADM, 195
 - list problems, 205, 208, 235, 294–299, 315, see also moding
 - models, see models
 - Notation Director, 216, 239
 - presentations with, 242–245
 - primary sources, 196
 - procedures, 204–205
 - Read-Write Variables, see RWVs
 - redefinitions, 207–209
 - RWVs, 200, 202–205, 207, 208
 - definition of, 199
 - distinction with FVs, 199
 - Sasami, 12_m, 224–229, 240, 251, 253
 - models, 227
 - scheduler, 195, 267, 270–276
 - SCOUT, see SCOUT
 - single-threaded, 205, 223, 258
 - steering wheel, 251
 - `tkeden`, 105, 108, 129, 146, 155–161, 213, 214, 216, 219, 220, 223, 224, 229, 231, 239, 245, 247, 249, 251, 255, 256, 265, 294, 326, 327, 345, 351
 - `todo` keyword, 219, 251, 256, 258
 - `ttyeden`, 196, 198, 199, 201, 205, 219, 231, 249, 265
 - usage, 222
 - USB, 250
 - virtual machine, 270, 271, 322
 - stack of, 271
- Eden
 - differentiated from EDEN, 194, 195
 - primary goals, 260
 - script
 - shown as script graph, 180
- EDEN Database Definition
 - Interpreter, see EDDI
- EFL, see Empiricist Framework for Learning

- EM, see Empirical Modelling
emhttpdWard1999, 1
 empirical investigation, 7
 Empirical Modelling
 and computation, 29
 and software development, 2, 19, 23
 artefacts to support, 12
 concepts, 5
 and LSD, 50
 applicability of, 42
 definition of activity, 6
 distinction between application
 and development, 19
 favourable aspects, 2
 meaning, 6, 8
 not a method, 352
 PhD theses, 18, 20
 principles, 19_m
 problematic issues, 2
 Empiricism, see Radical Empiricism
 Empiricist Framework for Learning, 7
 empublic archive, 1_m, 18, 20, 37, 108
 end-user programming, 35
 evaluation strategies, 33, 60, 237,
 265–267, 344, see also
 topological sort
 breadth-first, 237, 259, 264, 265
 concurrent, see concurrent
 definition maintenance
 data-driven, 60, 271
 demand-driven, 60, 109, 271
 depth-first, 237, 259, 264, 265
 evaluation/storage strategies, 60,
 63, 97, 105, 108, 109, 194,
 259, 281, 285
 determining organisation of
 thesis, 61
 for EDEN-like DM, 262
 for efficient calculations on lists,
 297
 on demand, 37
 rule of thumb, 259, 262
 use:redefinition ratio, 61
 Excel, see under spreadsheet
- eXtreme Programming, 21
- F** _____
F dependency function mapping, see
 under DAM machine
 \mathcal{F} set of functions, see under DAM
 machine
 fishbone diagram, 4
 folk-dance for GCD, 55
 Forms/3, 37
 functional
 dependency, see under
 dependency
 functional languages, 30, 34
 adaptive, 40
- G** _____
 Garnet, 39
 GCD, 55
 graph, directed acyclic, see script
 graph
- H** _____
 hardware, see under dependency
 hoc, 196–200, 204, 219, 235, 269
 HOD, 80, 101_m, 155, 179, 214_m, 241,
 275–276, 305, 310, 323, 325
 if, 148, 155, 164, 275, 306, 317,
 318, 347
 human computing, 210
 humour
 attempted, xi, 72_m, 93_m, 106–108,
 152_m, 374 (see meline under
 ‘b’), 379
 previously, 123_m
 lost, i–386
- I** _____
 identity
 rather than ‘identifier’, 99_m
 incremental computation, 40
 incremental construction, see under
 modelling
 indirection, see under reference
 indivisibility, 5, 7, 16, 26, 37, 65, 66,
 88, 97, 287, 288, 293, 349

- layers of, 274
 interaction, 10
 combining manual and
 automated, 90
 same techniques for construction
 and modification, 11, 19, 52
 Interactive Situation Models, 12, see
 also models
introtoempressentWard2002, 1
 invalid transition, 104, 106, 195, see
 also under ADM
 ISBL, see under EDDI
 ISMs, see Interactive Situation
 Models
- J** _____
- JaM machine API, 89
 and the DAM machine, 98
 concurrent update in, 287
jam2Cartwright2001, 260
 James, William, see Radical
 Empiricism
jugsBeynon1988, 243
- K** _____
- K* block of redefinitions, see under
 DAM machine
 Kay, Alan, see under spreadsheet
 Knuth, Donald, see under topological
 sort and DAM machine,
 BRA
krustyRoe2002, 235, 240
- L** _____
- Language for Specification and
 Description, see LSD
 notation
 level assignment, see script graph
 Linux, 1, 157, 158, 220, 251, 253
 literal values, 37
 LLDN, 99, 109
 characteristics, 99
 reason for abbreviation, 99_m
 symbolic influence, 306
 translating into, 101, 117
 LOGO, 235, 242
logoparserRoe2002, 235, 240
 Low-Level Definitive Notation, see
 LLDN
 LSD
 account, 47–52, 54
 animating, 59, 77
 behaviour interpretations, 59
 cf ADM scripts, 58–59
 digital watch, 16, 48_m
 external observer perspective,
 50, 58
 operational semantics, 47
 railway, 50, 66, 74, 77, 89
 vehicle cruise controller, 48_m
 agent, 48, 58, 66
 performing parallel action, 65
 three views of understanding,
 50, 81
 command list, 49, 59, 75
 derivate observables, 66
 distinguishing from ADM, 59, 89
 guards, 48, 54
 referring to authentic or
 perceived values?, 65
 notation, 47
 ‘protocol’ vs ‘privilege’, 48
 actions, see privileges
 derivate observables, 48
 handle observables, 48
 oracle observables, 48
 privileges, see under privileges
 state observables, 48
 perceived vs authentic values, 58
 privileges, 48
 as ‘can’, not ‘will’, 50, 54, 58
 operational semantics, 64
 synchronised commands, lack of,
 66
 LSD account
 ease of modification hypothesis,
 351
IsmpresentationWard2001, 1, 242

- M**
-
- Maintainer of Dynamic Dependencies, see MoDD
- make, 38, 40, 260, 261
- meaning, see also under Empirical Modelling
and requirements, 22
personal, distinction with other work, 41
provided by dependency, 23
to manage complexity, 23
- meaningful state, see under state
- Microsoft Excel, see under spreadsheet
- MoDD, 108, 123_m
- Model-View-Controller pattern, 39
- modelling, see also under Radical Empiricism
incremental, 10, 16, 21, 146, 210, 229, 233
motivations for, 7
of requirements, 21
- modelrailwayRose2004*, 255
- models
3doxoRoe2001, 227
agentparserBrown2001, 235
agentparserHarfield2003, 235–238
arcaBird1991, 212, 256
arcaWard2002, 212
backroomWard2002, 1
billiardsCarter1999, 227
billiardsYung1996, 220
by the author, 1
Care’s planimeter, 12
carparkingsimMcHale2003, 227, 251, 252
cat flap, 91
catflapWard1997, 1
central heating system, 351
claytontunnelSun1999, 216–218
cruisecontrolBridge1991, 214, 215
cubesymWong2001, 227
definitivedmWard2001, 1
digital watch, see under LSD, account
digital watch interface, 14
eddipTruong1996, 231, 235
eddipWard2000, 231
eddirTruong1996, 231
emhttpdWard1999, 1
introtoempresentWard2002, 1
jam2Cartwright2001, 260
jugsBeynon1988, 243
krustyRoe2002, 235, 240
logoparserRoe2002, 235, 240
lmpresentationWard2001, 1, 242
modelrailwayRose2004, 255
musiccorexptWard1999, 1
oasysprivilegesWard2000, 1
platonicssolidsBirch2001, 227
projecttimetableKeen2000, 14
pyramixRoe2001, 227
railway, see also under LSD, account
physical, 253
railway accident, see Clayton Tunnel
railwayYung1995, 255
room3dsasamiCarter1999, 227
roomviewerYung1991, 180, 181
rubiksCarter1999, 224, 226, 227
sandHarfield2003, 235
sandSocket1992, 229, 235
sasamiexamplesCarter1999, 227
sqleddiBeynon2001, 235
sqleddiWard2003, 1
texteditorYung1987, 201–207, 209
timetabling instrument, 14
vcgWard1999, 1, 180
vehicle cruise controller, 214, see also under LSD, account
- moding, 301–303, 305
- modularity
script graph as an alternative, 270
- musiccorexptWard1999*, 1

N

non-determinism, see under ADM
and UNITY

NoPumpG, 37

O

oasysprivilegesWard2000, 1

observable

definition of, 5

in EDEN and spreadsheet, 16

perceptible, 7_{fn}, 8, 10

observables

stability of values, 11

one agent, see 1-agent

OO, 24

cf ADM entities, 53

cf moding, 303

open source

methodology, 21

OpenGL, 224, 225, 227, 228

Oracle, see under database

P

P program store, see under ADM

parallel execution, see under ADM

partial order, 40, 180

Penguins, 37, 39

Phillips machine, 8

piano chord playing, 66

PIC microcontroller, 253

pipe, see under UNIX

pixel, see under state

PL/0, 235

planimeter, 12

platonicsolidsBirch2001, 227

PowerPoint, 242

Pro*C, 231

program

compare circuit, 28–29

comprehension, 24

distinction with data, 19

extending or composing, 28

projecttimetableKeen2000, 14

propagation of change

time taken, 30, 32, 104

pyramixRoe2001, 227

R

Radical Empiricism, 3, 7_{fn}

and abstraction, 23, 23_{fn}

and Empirical Modelling, 6

modelling and representation, 8,
14

William James, 3

Dyaks of Borneo, 306_{fn}

railwayYung1995, 255

RCS, 250

real-time, see under software

redefinition

and evaluation strategies, 60

block, 80, see also under DAM
machine, BRA

definition of, 16

history, 247–250

implying evaluation, see
evaluation strategies

in ADM, see under ADM

occurs at a point in time, 53

ordering significance, 67

redundant, 106

set, 262, 265

reductionism, see analytical reduction

reference

functional interpretation, 297,
305, 317

indirect, 33, 39, 154, 305

to subsets of state, 325

referent, 7

regular expressions, 235

requirements problem, 20

and meaning, 22

RISKS-LIST archive, 12, 20_{fn}

room3dsasamiCarter1999, 227

roomviewerYung1991, 180, 181

rubiksCarter1999, 224, 226, 227

S

S initial state, see under ADM

S' resultant state, see under ADM

S^* intermediate states, see under
ADM

- sandHarfield2003*, 235
sandSocket1992, 229, 235
 Sasami, see under EDEN
sasamiexamplesCarter1999, 227
 scalability, 34
 SCOUT, 194, 210–212, 214, 216, 219, 224, 227, 229, 237, 240, 247, 253
 model of `tkeden` interface, 247
 undue prominence in `tkeden`, 229
 SScreen LayOUT notation, see SCOUT
 script graph, 22, 27, 117, 178–193, 262–265
 ‘curtains’, 293
 ‘rays’, 294
 adding definition-agents, 281
 alternative to modularity, 270
 arc, 178
 changed by HODs, 275
 cycle, 67, 178, 287, see also under DAM machine, cyclic
 dependency
 ‘phantom’, 296
 cycles
 ‘phantom’, 317
 detection complicated by HOD, 326
 dependency structure, 184
 dynamically changing, 195
 extended, 282, 305, see also definition, translating to action
 isomorphic, 184
 level assignment, 179, 281, 285, 287
 modify internal detail, 233
 nodes, 178
 arcs per, 321
 decomposing, 285
 definition-agent, 282
 internal, 183
 isolated, 184
 leaves, 182
 roots, 183
 sinks, 183
 value, 282
 sources, see sources
 subgraphs, disjoint, 321
 subject to dependent change, 306
 targets, see targets
 transformed to isomorphic form, 233
 triggers, see triggers
 SDL, 47
 sequences of actions, see under actions
 must be rerun from the start, 26
 no clues to meaning, 26
 side effect, 200, 208, 261, 264, see also under DAM machine
 software, see also program
 Backus, John, 29–30
 Brooks, Frederick
 accident, 20
 essence, 20, 22
 complexity
 intrinsic and accidental, 22
 managed by meaning, 23
 dependable, 32, 349
 development, see modelling
 diagrammatic form, 22, 27
 real time, 32, 105
 real-time, 256
 software development, see also under Empirical Modelling
 and dependency, 38
 distinction with use, 19
 related PhD theses, 20
 sources, 102, 114, 116, 117, 119, 129, 137, 139, 148, 154, 155, 173, 174, 263, 269
 adjacent
 definition of, 184
 definition of, 184, 262
 preferred to ‘dependees’, 184
 preferred to ‘dependencies’, 102
 recursive
 definition of, 184
 speculative evaluation, 271, see also

- evaluation strategies,
 - data-driven
 - spreadsheet, 10, 40, 260
 - ‘generalised’, 162, 306
 - ‘instantaneous’ abstraction, 30
 - ‘prehistory’, 33
 - cf definitive script, 26, 32
 - comparison with EDEN, 16
 - errors, 36
 - Excel, 33, 36
 - reference style, 324
 - three-dimensional, 306
 - GNUmeric, 34
 - implementation, 33–35
 - Kay’s value rule, 34, 37
 - OpenOffice/StarOffice, 34
 - popularity, 35
 - properties, 36
 - scholarly publications, 34
 - testing, 36
 - VisiCalc, 10_m, 33
 - SQL, see under EDDI
 - sqleddiBeynon2001*, 235
 - sqleddiWard2003*, 1
 - SR, 289, 328
 - state
 - change, 6, 29
 - consistency within, 23
 - definitive, see under definitive
 - dirty/clean, 117, 119, 120, 131, 137, 139
 - global, 58
 - incorporated into dataflow
 - architecture, 40
 - intermediate, 28, 29
 - meaningful, 11, 23, 24, 26, 28, see also state, pixel, explanation
 - cf meaningful operations, 24, 25
 - prioritised over performance, 220
 - meaningless, 26
 - perceptible, 26
 - vs meaningful, 14
 - pixel, 110, 132, 166, 166_m, 173, 201, 205
 - explanation, 41, 98, 170, 174, 313
 - relationships between, 14
 - stable, 102
 - transition
 - in ADM, see under ADM
 - von Neumann, 42
 - vs behaviour, 52_m
 - steering wheel, see under EDEN
 - stimulus-response, 10, see cause and effect
- T** _____
- targets, 102, 103, 105–108, 137, 146, 169, 263, 269
 - adjacent
 - definition of, 184
 - definition of, 184, 262
 - preferred to ‘dependants’, 184
 - preferred to ‘dependents’, 102, 184
 - recursive
 - definition of, 184
 - telecommunications, 47
 - texteditorYung1987*, 201–207, 209
 - timetabling instrument, see *projecttimetableKeen2000*
 - topological sort, 108, 120, 180, 264
 - Knuth’s algorithm, 106, 180
 - transitions
 - major and minor, 30_m, 72, 104, 107, 108, 281, 287
 - tri-box framework, 303–327
 - definition of, 309
 - research questions, 307
 - topology, 310, 322
 - tri-box, 309
 - value box, 309
 - triggers, 269
 - definition of, 184, 262
 - typewriter, 42, 177
- U** _____
- UML, 22
 - UNITY, 81–88

- control flow, 84
- non-determinism, 84
- program execution, 84
- similarity with ADM, 85
- transparent variable, 86

UNIX

- pipe, 28, 210, 212, 216, 231, 256,
257

USB, see under EDEN

use-cases, 21

V

vcgWard1999, 1, 180*vi*, 201

VisiCalc, see under spreadsheet

von Neumann, John, see under
computerX

xeden, 214, 219

XP, see eXtreme Programming

xvcg, 180, 186

Colophon

This thesis was produced using an Apple iBook laptop running Mac OS X. Most of the text for this thesis was originally written in Microsoft Word version ‘X’, then manually converted to the \LaTeX text mark-up language. \TeX Shop was used as previewer and editor (along with TextEdit, `emacs` and SubEthaEdit). `pdftex` was the underlying typesetter. \LaTeX packages by various authors were used (with many thanks), including `amssymb`, `array`, `backrefx`, `color`, `extramarks`, `fancyhdr`, `fancyvrb`, `fix2col`, `fnbreak`, `graphicx`, `hyperref`, `ifthen`, `makeidx`, `pdfsync`, `setspace`, `shortvrb`, `showkeys`, `stmaryrd`, `url`, `verbatim`, `xspace` and a much-modified version of the Warwick `wnewthesis` thesis \LaTeX style file maintained by Mark Hadley. BibDesk was used to maintain bibliographic information that was later processed by Bib \TeX . `makeindex` was used to assist in typesetting the index. Most of the figures were produced using either OmniGraffle (for vector-based figures) or Adobe Photoshop Elements (for bitmaps). Donald Knuth’s Computer Modern fonts are used throughout.