Design Interfaces for High-Level Synthesis: Library Modelling, Netlist Generation and Visualisation

by

S S Johal

PhD

University of Edinburgh

1993

Abstract

In the fast growing field of high-level synthesis, very little attention has been paid to the areas where core synthesis tools must interact with their immediate environment. Library modelling, netlist generation and design visualisation are the three interfaces that have been neglected at the expense of advances in core synthesis tools. This thesis addresses this problem by looking at these primary interfaces and developing the ideas and tools that are needed to provide significant improvements over and above interfaces used by existing systems.

Most of the results of this work has been embodied in the development of the SAGE high-level synthesis system, whose most significant difference between existing high-level synthesis systems is that the electronic design engineer is able to direct the process of synthesis to a very fine degree of granularity. The main vehicle that has helped achieve this is the visibility of design information through graphical representations with which a designer is able to directly interact. This is in stark contrast to the purely automatic approaches of many synthesis systems, whose only support in heading towards the desired solution tends to be in the form of restarting a synthesis session from scratch.

As well as the interfaces themselves, support tools in the form of sound software building blocks combined with software frameworks around which solid interfaces can be built are equally important. Without them, the interfaces would be concepts without proof in reality. Consequently, an equally important problem that this thesis addresses is the development of the necessary tools that can ensure this can happen.

Credits

I would like to thank my supervisors Peter Denyer and Peter Grant for their help and support during the preparation of this thesis. I would also like to gratefully acknowledge the direct and indirect support from Abdul Sardharwalla, John Whitelegge, Mike Ginn, Nigel Baldwin, Phil Addison, Jonathon Puddicombe, Martin Ryder, David Mallon, Patrick Seymour, Colin Carruthers, Ross Kennedy, Douglas Chilsom, Allan Tomlinson, Irene Buchanan, David Rees, Tom Kean, Douglas Grant, Iain Findlay, Paul Neil, Gerald Allan and of course Magnus Magnuson.

This is in addition to many thanks to GEC-Marconi Research Centre and the Department of Electrical Engineering at Edinburgh University for supporting me during the development of this thesis.

Declaration

This thesis has been entirely composed by myself and except where indicated, the majority of the work presented in this thesis is my own.

Subindrao Singh Johal

June 1993

Table of Contents

Abstract ii

Dedication iii

Credits iv

Declaration iv

Table of Contents v

List of Figures vii

List of Abbreviations x

1. Introduction 1
    - 1.1 Problem Domain 4
    - 1.2 Thesis Outline 6
2. Background 9
    - 2.1 Terminology 9
    - 2.2 Research High-Level Synthesis Systems 10
        - 2.2.1 General 11
        - 2.2.2 Library Modelling Aspects 14
        - 2.2.3 Design Visualisation Aspects 15
        - 2.2.4 Netlist Generation Aspects 17
    - 2.3 Commercial Synthesis Systems 18
    - 2.4 Development History of SAGE 20
3. Library Modelling and Netlist Generation 21
    - 3.1 Behavioural Modelling 22
        - 3.1.1 Temporal Issues 22
        - 3.1.2 Library Modelling Language 25
            - 3.1.2.1 Combinatorial Objects 27
            - 3.1.2.2 Clocked Objects 29
            - 3.1.2.3 Pipelined Objects 33
    - 3.2 Structure Modelling 36
        - 3.2.1 Creation 37
        - 3.2.2 Attributing for Netlist Generation 44
        - 3.2.3 Generation 46
            - 3.2.3.1 Name Space Problems 49
            - 3.2.3.2 Generating ELLA 55
            - 3.2.3.3 Mapping Schematics 64
    - 3.3 Matchmaking and Unification 65
        - 3.3.1 Match Searching 65
        - 3.3.2 Unification 66
4. Design Visualisation 71
    - 4.1 Development of Visualisation 72
    - 4.2 Visuals Overview 75
    - 4.3 SAGE Data Model 77
    - 4.4 Visuals 80
        - 4.4.1 Graphs 80

4.4.1.1 Graph Categories 81
4.4.1.2 Invisibility 83
4.4.1.3 Nodes 86
4.4.1.4 Arcs 89
4.4.1.5 Constraints 89
4.4.2 Text 90
4.5 Drawing Directed and Undirected Graphs 92
4.5.1 String Method for Breaking Loops 93
4.5.2 Barycenter Application 96
4.5.3 Arrow Heads 97
4.6 Inter-Visual Interaction 100
4.7 Dynamic Visualisation 102
4.8 Visuals in Action 103
4.8.1 Source Code 104
4.8.2 Resource-Time Graphs 105
4.8.2.1 Zones 105
4.8.2.2 Call Shape 107
4.8.2.3 Mapping 112
4.8.3 Control Flow Graphs 113
4.8.4 Structure Graphs 114
4.9 Interaction 117
5. Framework 120
5.1 Background Decisions 120
5.2 Process Model 121
5.3 Language Bindings 129
5.4 Drawing Model 133
5.5 Attribute Management 139
5.6 User Interface Management Services 144
5.6.1 Programmers UI Interface 146
5.6.2 Rubberbanding 152
6. Foundations 155
6.1 Lists 157
6.2 Trees 161
6.3 Rectangle Management 162
6.4 Sparse Sets 168
6.5 Conversion Manager 171
6.6 Line Clipping 172
7. Results 176
7.1 Design Example 176
7.2 Critique 186
7.3 Future Work 191
8. Conclusion 194
References 197
Dissemination 204
Appendix 205

List of Figures

Figure 1-1. Typical SAGE Design Session ... 3
Figure 1-2. Major Interfaces to Synthesis Tools ... 5
Figure 1-3. Thesis Structure/Breakdown - Main Chapters ... 7
Figure 2-1. Major High-Level Synthesis Systems ... 12
Figure 3-1. Example of Modelling a Combinatorial Device ... 27
Figure 3-2. Setup, Hold and Propagation Model ... 30
Figure 3-3. Pipeline Model ... 31
Figure 3-4. Example of Modelling a Clocked Device ... 31
Figure 3-5. LML Example of a Clocked Device ... 33
Figure 3-6. Starting State for Pipeline Example ... 34
Figure 3-7. Pipeline Figure Combinations ... 35
Figure 3-8. Overall Context for Creation and Netlist Generation Activities ... 37
Figure 3-9. Creation Procedural Interface ... 43
Figure 3-10. Indirection and No-Indirection Comparison ... 45
Figure 3-11. Generic Outline for Indirection Support ... 46
Figure 3-12. ELLA names package ... 50
Figure 3-13. Name Mapping Example, Using Target Case Sensitivity and Case Directives ... 52
Figure 3-14. The 3 Databases Used in Unique Name Generation ... 52
Figure 3-15. Functions and Procedures present in Names Generic ... 53
Figure 3-16. Examples of Unique Name Generation ... 54
Figure 3-17. Names Database Management ... 55
Figure 3-18. ELLA Types ... 57
Figure 3-19. Netlist Generation Primitives: Buffer, Mux and Ram ... 58
Figure 3-20. Different ELLA Views of Tri-state Buffers ... 59
Figure 3-21. ELLA Bit Resolution Algorithm ... 59
Figure 3-22. The FNSET bit_bus Macro ... 60
Figure 3-23. Connection Types ... 61
Figure 3-24. IO -> IO Connections at Node and Instance Levels ... 61
Figure 3-25. Unused Inputs, Outputs and IOs Consumer ... 62
Figure 3-26. ELLA Controller MACRO ... 62
Figure 3-27. Generated ELLA Netlist Structure ... 63
Figure 3-28. Untouched Automatically Generated ELLA Example ... 63
Figure 3-29. Example of Leaf Level Mappings in Generated ELLA ... 64
Figure 3-30. Matchmaking Selection Panel ... 66
Figure 3-31. Unification Program ... 68
Figure 3-32. Unification Example ... 69
Figure 3-33. Run of Unification Example ... 70
Figure 4-1. Productivity vs Software vs Hardware ... 75

Figure 4-2. Design Loop 76
Figure 4-3. Basic SAGE Data Model Objects and their Interrelationships 79
Figure 4-4. Graph Categories 83
Figure 4-5. Examples Demonstrating Invisibility 85
Figure 4-6. Shapes and their Anchor Points 87
Figure 4-7. Pipelining Representation in SAGE 4, compared to in SAGE 2 88
Figure 4-8. Highlighting Text Items 92
Figure 4-9. Example to Illustrate the String Method 94
Figure 4-10. Space Allocation for Arcs 96
Figure 4-11. Transformations for Arrow Heads 98
Figure 4-12. Local Loops 99
Figure 4-13. Example Illustrating Hard and Soft Combined Axis 101
Figure 4-14. Dynamic Visualisation in Action 103
Figure 4-15. General Layout of the Source Code 104
Figure 4-16. Zone Identification Example 106
Figure 4-17. Zones in a Resource-Time Graph 107
Figure 4-18. Supported Call Classes 108
Figure 4-19. Call Classes and their Children 108
Figure 4-20. How a Call Looks Close Up 110
Figure 4-21. Arc Alignment, Depending on the Nature of the Call. 111
Figure 4-22. Mapping Example 113
Figure 4-23. Control Node Categories 114
Figure 4-24. The Three Grids for Schematic Routing (Pins, Components and Routing) 116
Figure 4-25. Navigation Panels 118
Figure 5-1. Stylised SAGE Process Model 122
Figure 5-2. Process Model - Normal Operation 123
Figure 5-3. Process Model - Debug/Test Operation 125
Figure 5-4. Process Model - Dual Process Operation 127
Figure 5-5. X Window System Conceptual Layers 129
Figure 5-6. PRAGMA and REPRESENTATION Clause Example 131
Figure 5-7. X-ADA Binding Scenarios 132
Figure 5-8. Rendering Model Overview 135
Figure 5-9. Composite Objects Through Hierarchical Construction 136
Figure 5-10. Region Mapping in Device and Object Spaces, for Non Orthogonal Mappings 137
Figure 5-11. Clip and Search Regions 138
Figure 5-12. Picture Attributes Example 141
Figure 5-13. Two Level Hashing Attribute Management 142
Figure 5-14. 36 Ways to Draw Text 143
Figure 5-15. Rotated Bitmap Text 144
Figure 5-16. User Interface Component Construction Example 146

Figure 5-17. Stylised Message Construction Example 148
Figure 5-18. Registering Software 149
Figure 5-19. Typical Output 151
Figure 5-20. Tracking, Inheritance and Limits for the x Ordinate 153
Figure 5-21. Rubberbanding Cross-Hairs and Box Examples - Points Only 154
Figure 6-1. List Forms and O(1) Symmetric List Combine 158
Figure 6-2. Multi View Stability 159
Figure 6-3. SAGE vs Booch iteration 160
Figure 6-4. Generics in C 161
Figure 6-5. Quad Tree Storage - Bisector Element Management 164
Figure 6-6. Waveform Culling Using The Rectangle Manager 167
Figure 6-7. Complex Conversion Management of Data References 171
Figure 6-8. Convert Generic 172
Figure 6-9. NCN Line Starting Points - Shown by Shading 173
Figure 6-10. SAGE Start and End Line Points 174
Figure 6-11. FLAT_STYLE Code Snippet For Line Clipping 175
Figure 7-1. VHDL Package Code 177
Figure 7-2. VHDL Process Code 178
Figure 7-3. Control Flow for TWOADD Example 179
Figure 7-4. Resource-Time Graphs of TWOADD Example During Manipulation 182
Figure 7-5. Structure Graphs of TWOADD Example During Manipulation 185
Figure A-1. The Lists Generic 207
Figure A-2. Red-Black Trees Generic 211
Figure A-3. Binary Trees Generic 213
Figure A-4. nary Trees Generic 215
Figure A-5. Rectangle Manager Generic 219
Figure A-6. Sets Generic 221
Figure A-7. Union Algorithm 224

List of Abbreviations

ALU	Arithmetic Logic Unit
ASAP	As Soon As Possible
AVS	Advanced Visualisation System
BITBLT	BIT Block Logic Transfer
BNF	Backus Naur Form
CAR	head of a list
CDR	tail of a list
DPI	Dots Per Inch
EDIF	Electronic Design Interchange Format
ELLA	ELectronic LAnguage
FSM	Finite State Machine
GC	Graphics Context
HDL	Hardware Description Language
HPGL	HPs Graphics Language
IPC	InterProcess Communication
LISP	LISt Processing language
LML	Library Modelling Language
MIMD	Multiple Instruction Multiple Data
OOD	Object Oriented Design
RCS	Revision Control System
PHIGS	Programmers Hierarchical Interactive Graphics System
POSIX	Portable Operating System Interface
SAIC	Science Applications International Corporation
SCCS	Source Code Control System
SIMD	Single Instruction Multiple Data
UIL	User Interface Language
UIMS	User Interface Management Services
VHDL	VHSIC Hardware Description Language
VTIP	VHDL Tool Integration Platform

LETTERS TO THE EDITOR (The Times of London)

Dear Sir,

I am firmly opposed to the spread of microchips either to the home or to the office. We have more than enough of them foisted upon us in public places. They are a disgusting Americanism, and can only result in the farmers being forced to grow smaller potatoes, which in turn will cause massive unemployment in the already severely depressed agricultural industry.

Yours faithfully,

Capt. Quinton D'Arcy, J. P.

Sevenoaks

1. Introduction

The later half of the twentieth century has seen explosive technological achievements in many areas of human endeavour. None so much as in the areas that have arisen from being associated with semiconductor technology. With a symbiotic relationship with software, there has developed a fierce self-feeding loop between hardware and software resulting in continuous improvement - which itself is accelerating. Just as important, if not more so, are the spectacular spin-off benefits which have touched nearly every aspect of mankind. Many examples of such benefits abound, from imaging systems available to physicians, to world wide communication networks that let men and machines communicate with ease and simplicity making the world a smaller place. The systems are as rich in their diversity as they are in their application.

The material presented in this thesis has aimed at being part of this self-feeding loop - focusing on the needs of digital electronic design engineers by exploring the tools and ideas that are needed to accelerate and improve the development of new hardware. The problem is simple, namely one of helping designers become more productive. The solution is complex and has started to appear in recent years as a kaleidoscope of many tools and ideas from a variety of university and industrial organisations. What all these developments have in common is the concept of high-level synthesis. In essence, this is the automation of converting concepts into silicon.

The main theme of this thesis is that of the major interfaces that a synthesis tool must have with its environment, and the foundations and framework on which such interfaces must exist in order to operate efficiently and effectively. Section 1.1 looks

at the problem domain addressed by this thesis and places these requirements in a much fuller context.

Most of the work presented in this thesis has arisen from work carried out on a three year collaborative research project hosted by Edinburgh University called SARI - Silicon Architectures Research Initiative. The project had several far-reaching aims, which included the development of tools and ideas in the area of high-level synthesis. The key feature that has differentiated the work of this program from the many previous efforts in the area of high-level synthesis, has been the direct inclusion of the human designer at every possible stage of the synthesis design route. This approach meant that all the creativity and ingenuity a designer possesses is married together efficiently and effectively with a spectrum of brute-force to clever synthesis algorithms. The main vehicle used to achieve this has been the use of graphical representations of a design as it passes from idea to implementation, guided by the human designer.

All the ideas developed have been embodied in one toolset, called SAGE [15]. SAGE stands for Sari Architecture GEnerator and as a tool, represents well over thirty man years of effort equating to over a quarter of a million lines of high level code. The toolset has been through a number of generations, the latest being SAGE 4. In its present from, the toolset represents an early snapshot of many ideas that are only now beginning to appear in more mature and stable environments in the form of commercially viable systems. The screen dump given in figure 1-1 shows a snapshot of a typical session with the SAGE system, illustrating some of the panels and graphs that a SAGE user sees and interacts with.

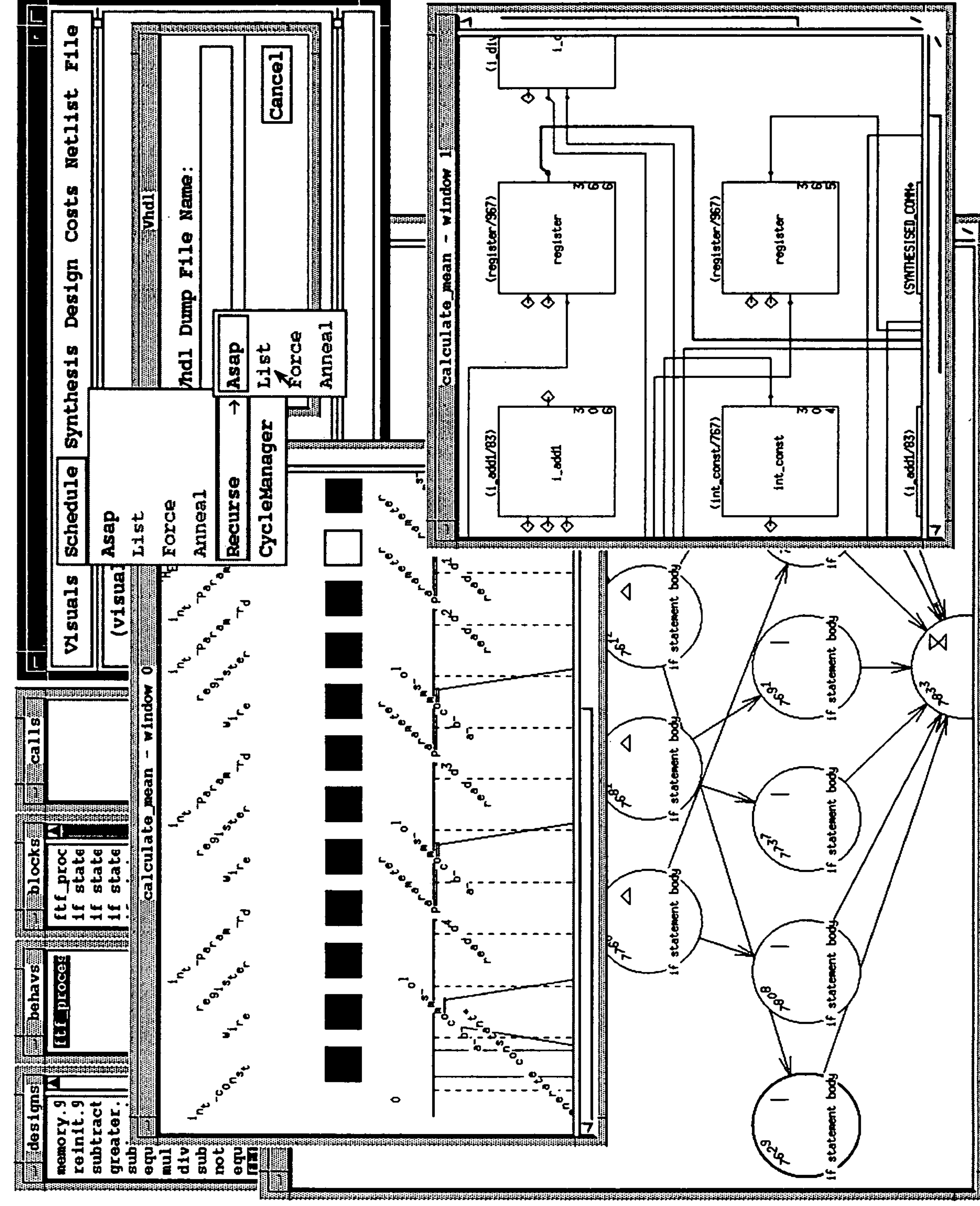


Figure 1-1. Typical SAGE *Design Session.*

The material presented in this thesis assumes a core understanding of several current systems. In particular, the ELLA hardware description language [34], the X Window System with its programming environment [57, 55] and the ADA programming language [3, 4, 5]. The following section starts by explaining in greater detail the problem domain that has been addressed by this thesis, and is then followed by section 1.2 which gives an overview of the material presented, and how it has been structured.

1.1 Problem Domain

High level synthesis compared with the problem of logic synthesis, usually involves starting with a high-level system description expressed in terms of data flow and control flow structures which precisely defines the system to be implemented. Such structures are similar to those found in most software languages, and have in recent years been incorporated into hardware description languages like VHDL [31, 32] and ELLA. Such descriptions involve time or temporal dependence, unlike the case with logic synthesis, where the focus is generally on purely combinatorial functions.

Figure 1-2 shows the high-level information flow of a highly stylised synthesis system. The structure of this diagram is representative of most synthesis systems. In simple terms, the overall order of operations starts with the input of source code and the target library information. Then, with the guidance of a user and the application of automatic core synthesis tools the aim is to produce a technology specific netlist as output. This thesis focuses on three of these four interfaces, namely library modelling, netlist generation and user interaction mostly aided by design visualization. Although a very important area, the problem of source code compilation has been largely obviated by the use of a commercial compilation system called VTIP [45]. In fact, this general approach of using existing tools to build upon, has been used wherever the existing tools have strongly satisfied the conditions of being either excellent or compliant with an existing or emerging industry standard.

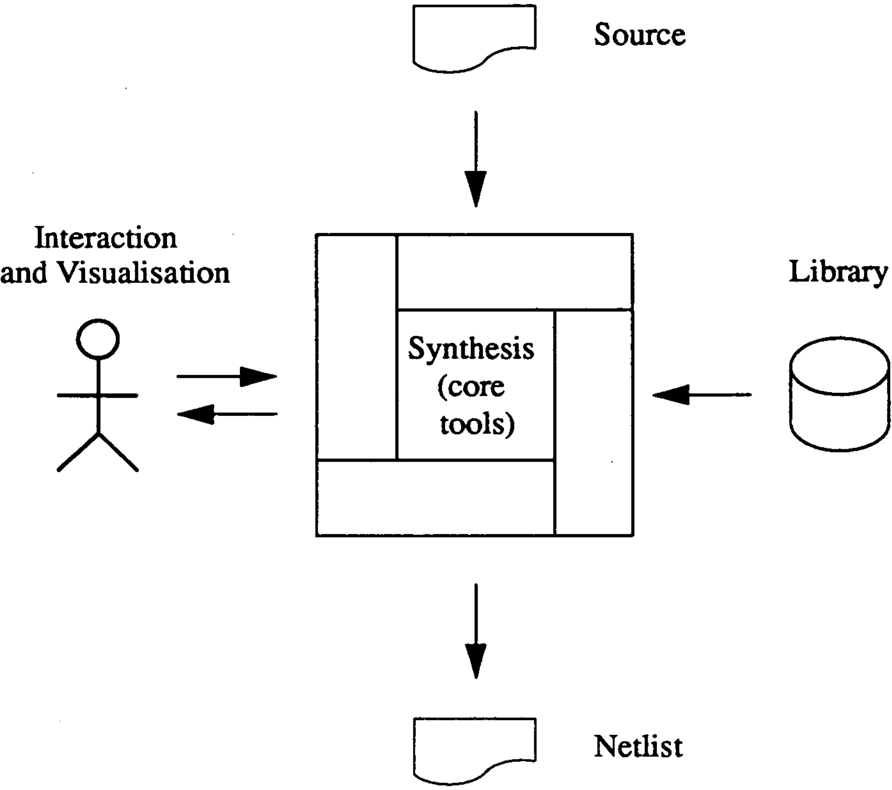


Figure 1-2. Major Interfaces to Synthesis Tools

Design automation tools to date have concentrated on empowering the user by providing highly focused or point tools that handle one or two concepts very well. Though this approach *does* provide solutions for designers, the tools tend to be too focused in their application. Additionally, this emphasis on the point tools has been at the expense of developments in the interfaces. In other words, this focus hides the richness of opportunity in carefully considered interfaces. By an examination of many existing interfaces, it becomes clear that a great deal more can be done to improve the integration of design automation tools within existing design flows, as well as meshing in closer with the natural requirements of a designer. The improvement of these interfaces is a large part of the problem area addressed by this thesis.

As well as the interfaces themselves, support tools in the form of sound software building blocks combined with software frameworks around which solid interfaces can be built are equally important. Without them, the interfaces would be concepts without proof in reality. Consequently, an equally important problem that has been

addressed by this thesis is the development of the necessary tools that can ensure this can happen. The aim has been towards general and complete solutions that could be used without loss of generality in many other novel tool development problems.

1.2 Thesis Outline

This thesis consists of eight chapters. After this introduction, chapter 2 begins with a detailed look at a large number of university developed high-level synthesis systems, and how they compare with the SAGE system, particularly with regards to the material presented in this thesis. The background also looks at the current state of play of commercial synthesis systems. With the multi-million pound research budgets that many commercially driven companies have had in recent years at their disposal, many of the technological developments are coming in larger and larger parts from within industry itself and therefore commercial systems are becoming much more representative of technological leaders in the field. The final part of the background chapter gives a historical perspective of the SAGE synthesis system, which helps place in context the ideas supported by the SAGE system as presented in this thesis.

Chapters 3 to 6 contain the core material presented in this thesis. The organisation and the inter-relationship of this material is illustrated pictorially in figure 1-3 by analogy with the core constituents of a building. Chapters 3 and 4, address the areas of design interfaces, showing several key new ideas that are in advance in many ways over existing approaches. Chapters 5 and 6, concentrate on what could be described as software structural foundations and frameworks respectively, without which many of the ideas in chapters 3 and 4 would not have been possible to explore or develop. With a practical basic approach, the broad contribution made by the material in chapters 5 and 6 has been that of an attention to thoroughness, completeness and accuracy in the development of enabling facilities and software. As with real buildings, the framework and foundations are very important and must be sound, since they form the support on which things must bind securely to be successful.

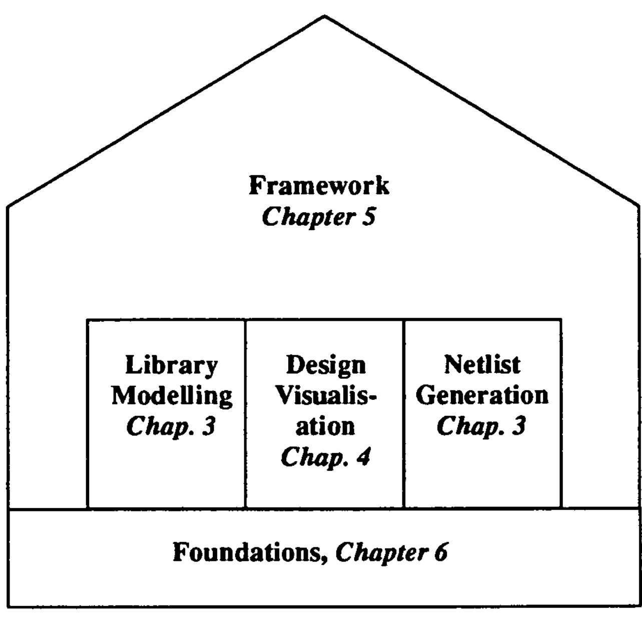


Figure 1-3. Thesis Structure/Breakdown - Main Chapters

Chapter 7, on the results takes a step back from the specific ideas presented and after describing a small design example using SAGE, goes critically onto examine the ideas that the SAGE system has tried to support. From this analysis is also presented the many additional ideas that could be further explored and developed. Although the thesis develops many ideas needed by electronic design engineers in the synthesis of complex systems, just as many important issues have been avoided. From testability issues to the special requirements of recently emerging sub-micron and very high speed logic systems, these holes are explored. In many ways, this chapter highlights the never ending room for improvement that always seems to be available in CAD tool development, and, more importantly, is on many occasions demanded by electronic design engineers. The final chapter, chapter 8, provides a summary of the material presented in this thesis.

The final few paragraphs in this introduction take a more detailed look at the material presented in chapters 3 to 6, with a view to highlighting the key contributions to the field of work that this thesis has made.

Chapter 3 addresses the problems of library modelling and netlist generation by showing the development of behavioural and structural modelling techniques based

on a formal language called LML. This language supports combinatorial, clocked and pipelined digital devices supporting two levels of timing representation. The structure aspects described in this chapter highlight the concept of creation supported by the technique of attributing netlists to help simplify the process of netlist generation. With these two concepts in place, the section on structure provides a comprehensive solution to the name space problem which is then used as the basis on which netlist generation of ELLA and schematics can happen. Chapter 3 finishes by looking at the services required by core synthesis tools when interfacing with a library of parts, namely matchmaking and unification.

Chapter 4 begins by placing in context the requirements of visualization and then exploring the taxonomy of graphical representations in the form of visuals which can help a designer. From an idealised treatment which views graphs and their properties, the subsequent sections look at how graphs can be effectively drawn. The chapter then explores inter-visual relationships and dynamic visualisation techniques. Following these sections that treat the subject of visualization in a general manner, the chapter looks at the specific development of visuals as used in the SAGE system. The chapter finishes by looking at how a designer is able to interact with these visuals.

Chapter 5 addresses five key areas that are found to be needed to support high-level synthesis in the context of high-performance graphics when the X Window System, ADA and Motif form the key building blocks that have to be used for sound reasons of tool portability and maintainability [97, 98]. These are the process model, language bindings, a rendering model, picture attribute management and user interface management services.

Chapter 6, is primarily a study of object centred software tools that have completeness, accuracy and ease of use as their primary objectives.

An architect's first work is apt to be spare and clean. He knows he doesn't know what he's doing, so he does it carefully and with great restraint.

As he designs the first work, frill after frill and embellishment after embellishment occur to him. These get stored away to be used "next time". Sooner or later the first system is finished, and the architect, with firm confidence and a demonstrated mastery of that class of systems, is ready to build a second system.

This second is the most dangerous system a man ever designs. When he does his third and later ones, his prior experiences will confirm each other as to the general characteristics of such systems, and their differences will identify those parts of his experience that are particular and not generalizable.

The general tendency is to over-design the second system, using all the ideas and frills that were cautiously sidetracked on the first one. The result, as Ovid says, is a "big pile".

-- Frederick Brooks, "The Mythical Man Month"

2. Background

This chapter provides an overview of previous work in the area of high-level synthesis. Of particular interest are the interfaces of such systems, and therefore these are examined closely, to help place in context the material presented in this thesis. The background material is in four sections. The first examines the key terminology used in the field. The second is a look at over twenty high-level synthesis systems primarily from universities. The third section examines the current state of the art in commercial synthesis systems. The final section is a historical perspective on the development of SAGE

Although high-level synthesis can be traced back to the 1960's [60, 61, 62], only in recent years have the ideas of high-level synthesis started to take hold as the promised productivity gains of such tools bear fruit. Not only are design times reduced to the order of days for complex systems, but also verification requirements are less due to correctness by construction and a designer is able to work closer to the problem and therefore concentrate on algorithmic problems rather than implementation problems. During this period, the terminology of high-level synthesis has also begun to settle.

2.1 Terminology

High-level synthesis, also known as behavioural synthesis, can be defined as translating a behavioural description into a controller description and a network of functional building blocks with their interconnections. The behavioural description is usually specified in terms of high-level operations. These operations are similar to the

data-flow and control-flow constructs found in software programming languages as well as in the behavioural parts of hardware description languages. Examples include assignment and arithmetic functions as data-flow operations, and 'for-loops', 'if-then-else' and sequencing as control-flow operations. Three broad stages are involved as part of the high-level synthesis process. Finding the functional building blocks to be used is called allocating. Assigning operations to functional building blocks is called binding. Assigning timeslots to each operation is called scheduling. In allocating, scheduling and binding, the target is to minimize one or more cost functions. Such cost functions are usually the area of the functional building blocks or the number of timeslots used. In the literature, the term scheduling is commonly used as the activity of applying all three of these stages optimally. The actual target hardware can range from general purpose functional units to specific hardwired datapaths that require a controller in the form of microinstructions.

Another commonly used term in the literature, is silicon compiler [84]. The broad definition of this term, fully encloses high-level synthesis, by being defined as a program, or set of programs that translates a behavioural description of a system into a chip layout. High-level synthesis, together with logic synthesis and layout form the three main stages of a silicon compiler. The dividing line between high-level synthesis and logic synthesis is not hard and fast. Nevertheless logic synthesis is generally understood to mean the synthesis of combinatorial multi-level or 2-level logic equations.

2.2 Research High-Level Synthesis Systems

The broad thrust of work presented in the literature has been that of heuristic solutions to the scheduling problem with relatively little attention to the interfaces that such a tool must provide. The library interface problem has been avoided by synthesis systems allocating complex objects directly. The most common example of this is the presence of the non-deterministic division operation as if it had the characteristics of a deterministic adder. Since most of the synthesis systems are targeted at specific internal logic synthesis and layout systems, the problem of general netlist generation is avoided and therefore this area, and particularly that of name space mapping in

netlist generation, has not been addressed in the literature. In the area of user interaction and visualisation, some recent work has started at trying to involve the user, but generally most systems are iterative rather than interactive in nature. The main exception is work done at Carnegie-Mellon University, which, while still no where near as flexible as SAGE, has focused on user directed behavioural transformations similar to those found in software compiler technology.

2.2.1 General

There are now well over thirty major high-level synthesis systems that have been developed at universities or research centres. Figure 2-1 contains a list of most of these synthesis systems with an indication of their origin and main developer. These systems form the basis of the following discussion. The systems which have not been

discussed are generally older systems like MACPITTS, whose functionality is more than addressed by the later systems.

System	Origin	Developer
ACE	Waterloo University, Canada	Buset
ADAM	University of Southern California, USA	Knapp
CATHEDRAL II	Inter University Micro Electronics Center, Belgium	De Man
CHIPPE	University of Illinois, USA	Pangrle
CMU-DA	Carnegie-Mellon University, USA	Thomas
DAA	Carnegie-Mellon University, USA	Kowalski
DSL	Karlshrue University, Germany	Camposano
DSS	University of Cincinnati, USA	Roy
DTAS	California University, USA	Dutt
FLAMEL	Stanford University, USA	Trickey
HAL	Carleton University, Canada	Paulin
HYPER	University of California at Berkeley (UCB), USA	Chu
LAGER	National Chiao Tung University, China & UCB, USA	Shung
LAMBDA	Brunel University, UK	Fourman
RLEXT	University of Illinois, USA	Knapp
MOVIE	Lund University, Sweden	Andersson
OCCAM TO SILICON	Meiko, UK	May
OLYMPUS	Stanford University, USA	DeMicheli
SEHWA	University of Southern California, USA	Park
SCHOLAR	University of Southampton, UK	Bergamaschi
SYSTEM ARCHITECT'S WORKBENCH	Carnegie-Mellon University, USA	Thomas
UCB'S SYSTEM	UCB, USA	Devadas
ULYSSES	Rutgers and Carnegie-Mellon Universities, USA	Bushnell
VSS	University of California at Irvine, USA	Lis
YSC	IBM T.J.Watson Research Centre, USA	Brayton

Figure 2-1. Major High-Level Synthesis Systems

The core synthesis activity of scheduling has a number of recognised approaches. ASAP (as soon as possible), ALAP (as late as possible), AFAP (as fast as possible), AEAP (as early as possible) are just some of the scheduling algorithms that can be

found at the heart of high-level synthesis. More advanced algorithms include list-scheduling, force-directed scheduling and simulated annealing. The force directed algorithm used by HAL [71], is one of the more interesting of these algorithms. It reduces the number of functional units, storage units and buses required by balancing the concurrency of operations assigned to them. The SAGE system has implemented variants of the ASAP and list scheduling algorithms. The UCB's synthesis system [88], is an example of a simulated annealing based design system. In this system, all the allocation sub problems, namely operator, memory and communication allocation, are tackled simultaneously. In SAGE, these stages are distinct and separate. FLAMEL [66] has shown how global optimisation at the control-flow graph level, can provide more benefit than simple local optimisation. The system does not support hierarchy and constrains multiplication and divide operations to powers of 2. The Yorktown Silicon Compiler - YSC [82], is a comprehensive design system that spans the full design process. It uses an APL-like language to describe the behaviour of systems. The Karlshrue DSL Synthesis System [69], has a purpose-built input language that supports applicative and imperative description styles. The synthesis algorithms use both global and local automatic optimisation before mapping to real hardware components using parametric based module generators. Most of these systems represent general approaches to the high-level synthesis problem.

A number of synthesis systems have focused on just one part of the high-level synthesis problem. SHEWA - a Korean name meaning 'flower of the world' [68], is targeted explicitly at pipeline synthesis. Using brute-force methods, an exhaustive design exploration algorithm ensures that the minimum cost, highest performance design can always be found. In comparison, SAGE has been designed so that it can be guided directly by a user to the right solution.

Most high-level synthesis tools explore the design space and present the user with a range of results or the best result according to the constraints imposed by the user on the synthesis process. The DSS - Distributed Synthesis System [89], has exploited this feature by being targeted at MIMD computer architectures. In many ways, SAGE has the correct process model to support the same concepts (as discussed in section 5.2

on page 121), but currently has not been developed to the same extent as DSS. DSS also supports VHDL [31] as its specification language.

2.2.2 Library Modelling Aspects

Rather like SAGE's matchmaking facilities, the DTAS tool [90], is able to map technology independent libraries to register transfer level libraries. This is in comparison with most other synthesis tools, where functional units are passed onto combinatorial synthesis tools like ESPRESSO. The libraries are specified in a language called LEGEND which includes semantics for clocking, asynchronous behaviour, bidirectionals and generic components. The description is then mapped into GENUS, which generates the generics, and forms the library database that DTAS uses. Unlike SAGE, the approach is rule based and does not support general cost attributes and low level timing information.

The SCHOLAR [80] system is another example of a general high-level synthesis tool, but with an interesting library mapping backend. It uses a logic synthesis program called SKOL [79], which has a fast technology mapping algorithm to map functional units to a target library. The approach is based on the use of a numerical string for representing the boolean expressions and library cells, which contributes to the fast selection process. Another performance enhancing feature is that the boolean expressions do not need to be decomposed to primitive gates before the mapping process.

LAGER [77], is typical of a number of systems that achieve technology mapping by having a well defined target architecture. The target architecture is a parameterised structure, and the result of mapping is to produce microcode and parameter values. If, after synthesis the user is unhappy with the result, the user can select another target architecture and restart the synthesis. LAGER uses SILAGE as well as a C-like [8] input language. A similar approach has been taken in [91], where OCCAM is mapped onto multiple abstract micro-machines.

2.2.3 Design Visualisation Aspects

The OLYMPUS [92] system allows limited interaction, to the extent of in-lining procedures calls or explicitly specifying which library modules to use for specific components. Automatic behavioural transforms include constant and variable folding, common subexpression elimination, dead code removal, reduction of constant conditionals and loop unrolling.

VSS (VHDL Synthesis System) [93], is one of the few systems that support synthesis from VHDL and allows limited visualisation facilities in the form of viewing internal flow graphs after the compilation process. The system also includes automatic behavioural transformations that support loop pipelining as opposed to loop unrolling which it is also able to support.

Using 'knobs' and 'gauges' as graphical metaphors, the CHIPPE [74, 67] system provides a user with an organised way to control the high-level synthesis process. The design process involves setting the user constraints (knobs) and analysing the results (gauges) and then iterating around this loop. The gauges of quality include measures for area, power and time, as well as overlap (the number of states for which two units are active in parallel), dead time (how much the system clock is not used) and bus usage. Although a designer has greater visibility of the synthesised designs quality, the process is still iterative rather than interactive.

The IBA system [86] provides a designer with interactive design transformations on a synthesised design. The system has two major problems. Firstly, it is textually based and therefore a user has no immediate insight as to the effect of the transforms. Secondly, the designer is unconstrained as to which transforms to apply. This allows the normal paradigm of correct by construction to be violated. This arises because the cleverness of the IBA system is in the RLEXT program which can 'fix' a design once a user has modified it. The range of transformations include addition and removal of functional units, as well as control step schedule modifications by shifting operations, inserting or deleting control steps and compressing sequences of control steps.

Through the use of formal proofs, the LAMDA (Logic And Mathematics Behind Design Automation) [63], system is able to be guided by the user to an implementation. Since formal textual equations drive the synthesis process, the process is initially non-intuitive for engineers and lacks the graphical expressivity found in SAGE visuals.

Some systems appear from a first glance to provide user interaction and design visualisation facilities similar to SAGE, but in fact do not. One example is HYPER [85]. Unfortunately, the term interactive is confused with the more usual iterative facilities provided by existing high-level synthesis systems. Thus a user can specify hardware, memory, connectivity and timing requirements which direct the actions of the automatic transformation and optimisation tools. An interesting point about the HYPER tool, is the extensions to the SILAGE input language. SILAGE is usually a DSP style applicative language, but the extensions add 'while' loop, 'if-then-else' and interprocessor communication constructs. Another system that claims to provide interactive facilities is MOVIE [72]. This is a full silicon compiler, which does provide significant interactive and high performance graphics at the layout stage (achieved by using dedicated graphics hardware), but only the necessary hooks for future high-level synthesis tools that might support the concept of interaction and visualisation. ULYSSES (Unified LaYout Specification and Simulation Environment for Silicon) [70], claims to let the designer interrupt the design process at any stage and take over control, but the granularity of that interruption is a function of the tools it is interfaced to. In addition the tools that have been integrated with ULYSSES, are again generally layout tools, where the granularity of the interaction is much more clearly defined. While the ACE [94] system methodology clearly recognises the difference between interactive as opposed to iterative synthesis, the current tool only lets a designer screen suggested transformations. The main innovation of interaction is in the graphical interactive specification process as opposed to a textual based language. The other main difference from SAGE is that the tool only supports data flow.

The CATHEDRAL II system [83] (like LAGER), is targeted explicitly at multiprocessor DSP systems. The target architecture is a set of concurrent dedicated bit-parallel processors on a single chip. The system uses SILAGE and through the use

of pragmas, is able to provide a limited degree of user control on the synthesis route. This was a formal design decision in that user 'interference' was to be limited as much as possible.

The USC Advanced Design AutoMation (ADAM) [78] system is designed to unify a number of design automation programs into a single framework including a knowledge based synthesis system. The system is in two parts, a planning engine (called DPE), and an estimation engine that evaluates a plan. The overall approach is modelled on how real designers do digital system design. Thus, if the 'execution' of the plan fails then control is passed back to the planning engine. The knowledge base of the planner is populated with register-transfer level concepts for system level digital design; it can also be populated with other knowledge sets.

Several systems have been developed at Carnegie-Mellon University. This includes the DAA - Design Automation Assistant [81], a rule based system. The work done at Carnegie-Mellon University now represents some of the most mature high-level synthesis systems developed to date. The current system, the SYSTEM ARCHITECTS WORKBENCH [87], forms a framework on which the high-level synthesis components such as CMU-DA [65, 73] have been built. Using ISPS (Instruction Set Processor Specification) as the input language, the system maps this to internal value-trace graphs which bridge the behavioural and structural domains. In a similar manner to SAGE, the three domains of input language, internal data+control flow and output structure are all linked graphically so that a designer can view the interrelationship of selected objects across all three domains. Of all the systems reviewed, this system provides the closest functionality to SAGE, in that it supports user directed transformations that allows the exploration of different architectures. Unlike SAGE, since the transformations are applied to the value-trace graphs, the effect of each transformation on the resulting data-path and controller can be difficult to judge.

2.2.4 Netlist Generation Aspects

While this survey of high-level synthesis systems has seen several different approaches to design visualisation as well as library modelling this has not been the

case with netlist generation. No particularly special approach has been adopted in any of these systems. Most are targeted at their internal netlist language and consequently have not had to handle the requirements of general and third-party netlist generation problem. Another reason for the lack of development of this interface is the infrequency of usage because of the general life cycle of research synthesis systems. That life cycle means that problems are remedied using simple workarounds rather than anything more elaborate.

2.3 Commercial Synthesis Systems

Companies involved exclusively in the development of CAD systems have grown at an explosive rate over the last ten years. Two of the largest companies are Mentor Graphics and Cadence, which both provide a comprehensive array of CAD tools that can help designers in a wide range of areas from system specification to polygon pushing. Only in the last few years have they started to address the areas of high-level synthesis. In many ways, the large part of their development effort has been expended on framework developments, providing the right system backbone for the current and future tools to coexist and work effectively together. It is interesting to note that Mentor Graphics spent over 100 million dollars over a three year period to migrate all their tools into their falcon framework, producing a system with well over 11 million lines of source code. Work by the CAD Framework Initiative (CFI) [64], has been directed at making different CAD systems work together. Current demonstrations have shown the transparent passing of netlists between several different systems' schematic capture packages. Future work will enhance the specification even further so that customers can pick and choose the best tools according to their needs from all the CAD tool vendors, and be able to integrate them seamlessley together as if they were purchased from only one vendor. The VTIP [45] is an example of an existing commercial package whose sole purpose is to act as a framework or platform on which to build VHDL applications, and is the main reason why it is used by SAGE. It achieves its function through the use of a the SPI (software procedural interface) [46], applied to design units organised by the DLS (Design Library System) [47]. What makes it even more interesting, is the 'views' that application developers using these tools can have of the VHDL database. These include 'compiler views', and

recently a 'synthesis view'. Use of such a synthesis view and general framework could have considerably helped the development of SAGE by supporting the tool design problem from a higher starting level of abstraction.

The development of the commercial synthesis tools has drawn heavily on the systems developed by universities. Formally, they are currently closer to logic synthesis than high-level synthesis in functionality. The Mentor Graphics AUTOLOGIC [40] toolset has parts of the CATHEDRAL work. The Cadence system makes use of MIS, BOLD, SIS and CADOPT [38], the latter being from an internal company development. The systems use VHDL and VERILOG [36] respectively as their primary input languages, but can also rework existing designs presented as EDIF [33] netlists or schematics. Whereas the Mentor Graphics system produces a straightforward synthesis run, the Cadence system provides a two stage synthesis process, the first of which gives sampled space-time graphs, from which the designer can select a particular implementation to run to completion. These tools are being continually developed, and many of the ideas of CASE tools are now being propagated into the hands of digital system designers. One of the latest offerings from Mentor Graphics is the System Design Station [44]. This helps in systems requirements capture with a direct path available to the AUTOLOGIC synthesis tools. Another emerging system is ExpressVHDL [43], which lets designers capture system requirements graphically using an extended form of state transition diagrams. From this, behavioural level VHDL can be produced which can be passed directly onto a synthesis system.

The SYNOPSIS [41] synthesis system has become the market leader by having exploited the current demand in industry for synthesis tools. Supporting synthesis from both VERILOG and VHDL, the toolset has developed a solid reputation through reliability and the addressing of additional synthesis problems including test synthesis. LOCAM [39] can use ELLA [34] or VHDL as its input language. While fast and efficient, the recent addition of optimisation across register boundaries by using the technique of register propagation distinguishes this tool from the others. Probably the closest in similarity to SAGE, is the Silicon 1076 [42] environment from LSI Logic. The system is able to produce control and data flow graphs that depict the

degree of parallelism in a design, but only in an iterative manner. The tool supports similar design exploration facilities to the Cadence synthesis system.

2.4 Development History of SAGE

To help place in context some of the developments outlined in this thesis, this section provides a brief historical overview of the development of SAGE. As mentioned in the introduction, the SAGE system was developed as part of the SARI - Silicon Architectures Research Initiative. This was one of a number of Department of Trade and Industry sponsored initiatives under the common title of NERI - National Electronic Research Initiative. The overall aim of these programs, including SARI, was to do pre-competitive research in newly emerging technologies. Other NERIs were associated with pattern recognition (RIPR - Research Into Pattern Recognition) and hybrid development (RISH - Research Into Silicon Hybrids).

Three broad phases were involved in the development of SAGE. Firstly, the philosophy development which identified the need to focus on flexibility, interaction and correctness as the three main features of SAGE [13, 14]. Secondly, the development of the SAGE 2 toolset [16, 17, 12]. It is with this toolset that the library modelling and netlist generation facilities discussed in this thesis were primarily associated [21, 18, 19, 20]. The third phase involved the development of SAGE 3 [15], which with a few minor additions is now called SAGE 4. The primary improvements over SAGE 2 included a more comprehensive data model that could support control-flow and hierarchy. The work relating to design visualisation and user interaction described in this thesis is related to this version of SAGE [22, 23, 24].

THE LESSER-KNOWN PROGRAMMING LANGUAGES #8: LAIDBACK

This language was developed at the Marin County Center for T'ai Chi, Mellowness and Computer Programming (now defunct), as an alternative to the more intense atmosphere in nearby Silicon Valley.

The center was ideal for programmers who liked to soak in hot tubs while they worked. Unfortunately few programmers could survive there because the center outlawed Pizza and Coca-Cola in favor of Tofu and Perrier.

Many mourn the demise of LAIDBACK because of its reputation as a gentle and non-threatening language since all error messages are in lower case. For example, LAIDBACK responded to syntax errors with the message:

"i hate to bother you, but i just can't relate to that. can you find the time to try it again?"

3. Library Modelling and Netlist Generation

This chapter investigates the problem of modelling behaviour and structure in a form that is suitable for high-level synthesis. Having defined structural objects, the problems associated with generating netlists are examined in section 3.2. The chapter finishes by looking at how a database of behavioural and structural objects can be used for the purposes of matchmaking and unification.

The distinction between behaviour and structure is not arbitrary. Strictly, the definition of behavioural models does include the case of structural models, but because of the dominating influence of hierarchy when considering structural components, their separate treatment from behavioural components can be considered valid.

Similarly, there is a subtle distinction between matchmaking and unification that tends to mirror the behaviour/structural divide. Matchmaking is the process of finding a suitable implementation for a given behaviour object, while unification looks at the problem of confirming the equivalence of two structural objects. It should be noted that this distinction is used within this thesis but, within the common literature, no such clear distinction currently exists.

3.1 Behavioural Modelling

As we have seen in the literature review, there are many different ways to capture the behaviour of devices. In addition to many methods to specify the same level of abstraction, there are many different levels of abstraction that can be specified. These different levels of abstraction start with highly focused tools that try to model reality in the greatest possible detail like SPICE [35]. At the other end of this range of levels are general all encompassing tools like BLM [37] (Behavioural Language Modelling in C/PASCAL), which provide modelling support for what is required in high-level specification terms, as opposed to the final implementation.

As examples to illustrate the variety of modelling approaches currently in use, we can see at one extreme for simulation purposes the advanced quickpart technology [48] in use by Mentor Graphics Corporation. At the other extreme is the technology used by synthesis systems like LOCAM [39] which allow a designer to describe the essential logic behaviour of behavioural primitives such as boolean equations.

To support this wide variety of descriptions, a behavioural description can be analysed in two basic parts. There is the temporal view point concerned with when and where a signal and operation must happen. The other strand is related to the actual behaviour without concern to time. The rationale that helps justify this distinction is simple. Timeless behaviour is generally associated with procedural steps that encapsulate the operation being executed as a series of data dependent relationships.

3.1.1 Temporal Issues

There are two general styles of describing the temporal behaviour of a system. These two styles are commonly termed asynchronous and synchronous. The main difference between the two styles, is the way that transfer of state information is governed by an explicit clock signal for synchronous logic. Normal design techniques usually allow asynchronous logic to be designed by identifying the feedback loops and then inserting fictitious registers or delay elements to allow the design to proceed. There are several additional checks that must be made for asynchronous logic, most notably checking for delay hazards. Because of these additional complications this thesis only

addresses the problem of designing with synchronous logic. Asynchronous logic can be treated in a similar fashion to synchronous logic, by the treatment of the inserted delay elements as clocked registers. This would mean that the problem of the asynchronous timing would become one of deciding on a suitable time resolution.

Many levels of temporal abstraction can be identified for logic based systems. The reflection of state is the most important. For stateless systems, the timing can be expressed as being zero time, or looked at as the relevant pin to pin timing. This modelling can then be broken down one further stage, to reflect the rise and fall delays as well as the value dependent timing paths. The most dominating value is the propagation delay, and therefore, most designs can be expressed by providing the maximum value of all the pin to pin delays. Adopting this option, is not to ignore the importance of the shortest delay path, which can cause problems in validating complex timing relationships. A classic example of where this is important is that of ensuring a minimum delay value between registers to prevent clock skew problems affecting design intention.

For architectural synthesis, a careful examination of the requirements can be used to identify the key first order effects that must be supported. What could be termed second order effects or higher can be fairly passed on to the point tools that can handle such detail. From the specification, the requirements are mapped to sequenced operations. Many operations are high level processes like multiply and divide, which will usually require a complex FSM to operate. There are also many lower level operations like add and subtract which can be mapped directly to a library of parts, whether in an ASIC library or TTL data book. The importance of the right first order effects being supported can be highlighted by exploring what would if happen if the wrong choices are made. In this case, after architectural synthesis, the next tool set needs to operate with minimal need to restart the synthesis process. A good example is if combinatorial components are treated as point (or zero time) operations, in which case no control of clock period duration can be made, and therefore, even though the resulting architecture will be logically correct, the performance in MHz will be unpredictable, since all operation elements were treated with the same point time behaviour.

The process of choosing which time parameters dominate and therefore are a necessary part in the modelled library components can be easily explained. With a single clock synchronous implementation methodology, the modelling of sequenced events is very important. This implies the use of a synchronising signal (i.e. a clock) and data-flow. For combinatorial objects, there are many levels of abstractions to choose from. At the highest level, as explained earlier, there is the treatment as point operations, effectively taking no time. Next we have the general propagation delay model, that is summary values for all pin to pin delays. The next level is to supply explicit pin to pin delays as well as to provide rise and fall figures. At this stage, the effect of capacitance and track loading can become important. Given these choices, in the category of first order effects is the general propagation figure since it has a direct bearing on the performance of a system.

With this decision, there is a direct need to look at the similar dominating delay figures associated with memory elements. The key points here are the need to ensure that the input is valid when a clock edge appears, and how long it takes for an output to settle once a clock event arrives. This abstraction is achieved through the use of delays that are commonly referred to as the setup, hold and propagation figures.

In a similar way to combinatorial elements, arguments for propagation, setup and hold values as providing sufficiently descriptive timing attributes for architectural level design can be made. The main argument is one of maintaining simplicity, such that more detailed representations as found in data books could be added at a later date if required. Consequently the main memory element is that of a `DTYPE` register element, defined as an edge sensitive device that can be stretched as appropriate to the required number of bits. For these same reasons, latches or level sensitive devices are not considered, because of the way they violate the synchronous design approach.

Even with this simplified memory model, the differences between the ideal memory element and those found in data sheets is worth noting, to ensure that it is an adequate and practical abstraction. As with so many issues related to synthesis, it is not the major mode of operation that is of concern, but details relating to the memory initialisation. Thus, even when memory elements come with synchronous and

asynchronous reset, it is generally easier to state that the algorithm is responsible for initialisation of the register in a synchronous fashion. This approach means that for non-trivial memory declarations that have to map to real hardware, the corresponding non-trivial initialisation algorithm can be incorporated in the algorithm. This is even more important when one considers issues relating to test, where specialist algorithms need to be developed.

When treated together, the temporal models for combinatorial and memory elements provide an adequate framework on which the three general styles of synchronous models of chaining, multicycling and pipelining can be supported. Note, even though such features are possible, their usage depends on the form of the synthesis tools that make use of these concepts. For example, no support for multicycling is made by the current generation of SAGE scheduling tools.

The main point about separating timing from behaviour for library models, is that of being able to support matchmaking in an efficient manner. The language that has been developed for SAGE to support this activity is described in the following section. Note though, in the historical development of SAGE, this language applied only to SAGE 2.

3.1.2 Library Modelling Language

The process of describing leaf level objects that can accurately encapsulate the requirements as described in the earlier sections, can be achieved through the definition of a library modelling language, henceforth referred to as LML. The approach requires that given any hardware object, a systematic list of that object's capabilities are made. A simple example is that of an ALU, where distinct capabilities are identified as ADD, SUBTRACT, MULTIPLY etc. - usually identified in data sheets with the exact state of control lines that are required to achieve the required function. A more involved example would be that of a DTYPE, where the usual capability of latching data, would be supplemented with the relatively more complex requirements of being able to reset or set the DTYPE, synchronously or asynchronously - depending on what the real life device supported.

The function itself might be decomposable into much smaller building blocks, for example an adder being composed of 'half' and 'full adders' to achieve the required function. The key point, is that though this may happen, the function is identified as distinct from its temporal behaviour. This means that the separate capabilities are identified by distinct timing views.

The level of detail in the function and timing views can be considerable. For the function, the major effects are signal structure, state dependence and operational semantics. With signal structure, many features such as signed/unsigned number representation, bit width and most significant bit designation are the key characteristics. State dependence can be supported to a limited extent, in as much as local state information can be represented in the timing view, illustrating a duality between the two forms of state representation. Clearly eliminating state dependence from a function simplifies it considerably. The final feature of operational semantics, which could be fairly argued as encompassing the previous features and effects, is the procedural statement of how a particular action happens.

The classic DTYPE is again a good example of this distinction. Here the signal structure can be anything. In fact modern languages now support polymorphic types that allow such expressive expression of inputs - as in languages like ELLA. Thus the signal structure can be anything, but with the state dependence furnished by a timing view, the operational semantics are as simple as Q = D.

The next few sections explore, by way of simple examples, how LML can be used successfully to describe a range of devices, in a form that captures the key details as required by the matchmaking process.

A few general comments about LML are required. LML is based on ADA [3], reflecting the verbose but expressive nature of that language. The language is case-insensitive and free format and the grammar is structured to organise related information in a user-friendly a way as possible. The final point concerns the minimization of ambiguity within the language, which is a necessary perquisite to help automatic tools that might be developed at a later date to produce automatically LML code.

3.1.2.1 Combinatorial Objects

The following example shown in figure 3-1, is that for a combinatorial device that has a propagation delay of 35ns. The timing diagram indicates the behaviour that is being modelled.

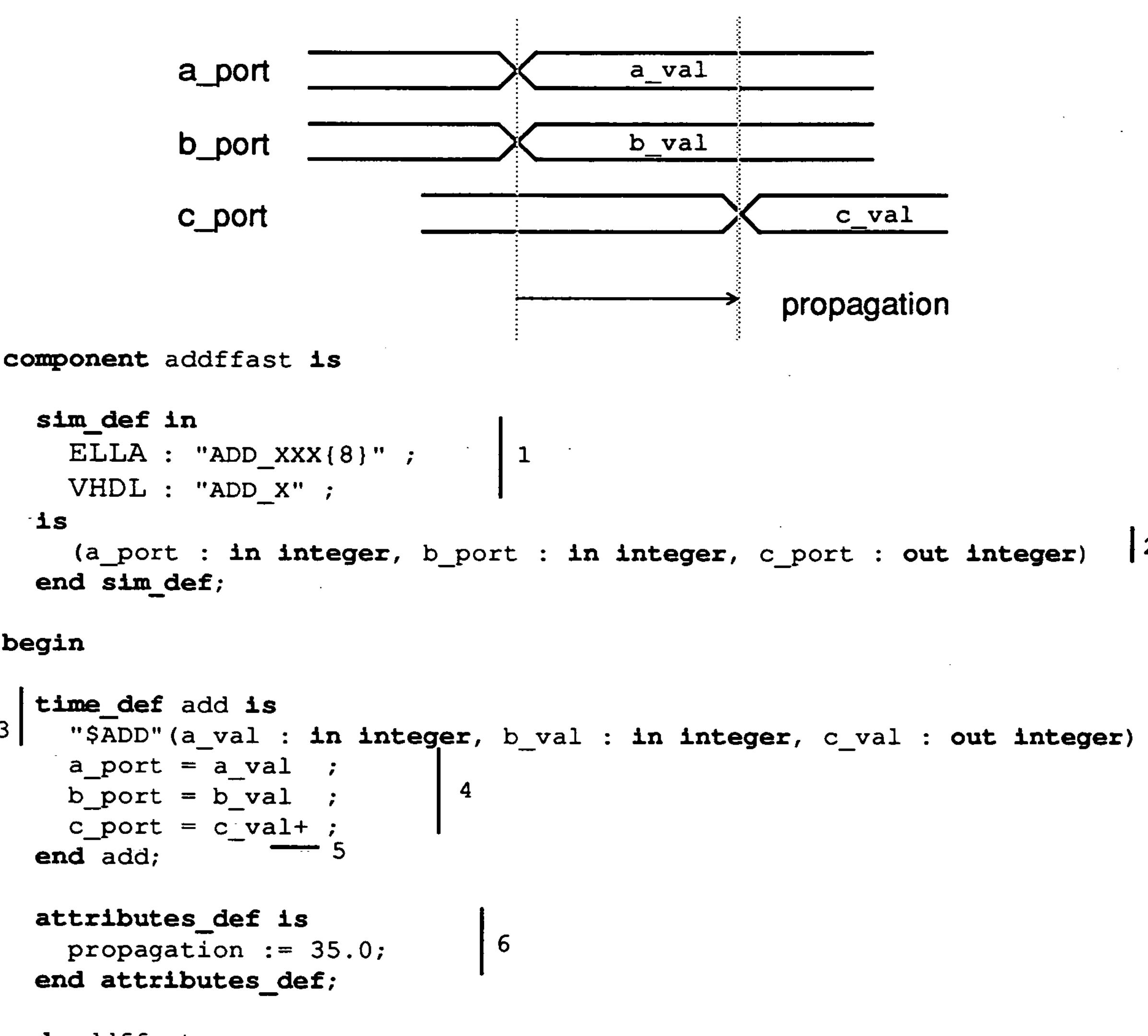


Figure 3-1. Example of Modelling a Combinatorial Device

With the wide variety of different hardware description languages that are now available, it is clearly important that a mechanism to support mapping between a LML name to a target library name is provided. LML achieves this by understanding the existence of ELLA, VHDL and EDIF [33] languages, while any unsupported language

can still be indicated by supplying the language within quotation marks. By providing hard coded language names, the compilation software provides two facilities. Firstly it is able to check errors within the name syntax at the earliest possible stage, and secondly many of the difficulties of managing name spaces can be eliminated at this stage. (How these name space problems are managed is discussed in section 3.2.3.1).

This information is contained within the simulation definition part, marked as '1' in the figure. It is included in what would be the declarative part of the equivalent ADA, since it is just that - a declaration. As a fallback option, the semantics imply that a target HDL that does not exist results in the default name being that which follows the 'component' keyword.

Part '2', which is common to all the different names that the unit can take on, is the formal parameter list. This list is the perspective used by the netlist generation phase, since it will contain all the physical signals associated with the component. As will be seen for some of the more complex clocked examples described later, it is possible to have a different set of signals describe the timing views. An analysis of all the signal types that appear in real systems, indicates that the three typical directions of inputs, outputs and bidirectional ports form a complete enough set useful for real designs.

This example consists of only one timing view as shown in '3'. It is formed from basically two parts, the first being the software function of the object and the second being a timing matrix that maps software values onto real ports. In this example it is an addition operation. The data flow of the object is to receive two values and produce a single result. The software operation, for matchmaking purposes, must be a known class of operation. This implies that all the classes of arithmetic operations found in a normal programming language are candidates to be declared as software operations. The situation is not as simple as this, and is explored more fully in section 3.3 where matchmaking is discussed.

In order to map software values to real ports, the concept of a timing matrix is used. The key feature of this concept is the mapping of events on software objects to values on ports. Part '4' of figure 3-1 shows how the mapping of these software values happens through the associated timing matrix. Each line should be read as the

mappings associated with a particular port. Thus the first line represents the value a_val on a_port, while the third line represents the generation of the c_val from the port c_port. The construction of these software events into columns is very important, since the event-action sequence is carried within this abstraction. As shown in part '5', any activity on ports that is an output activity is represented by the '+' symbol. In essence it is saying that on any events within this column (i.e. the arrival of a_val and/or b_val), then generate c_val after any delay requirements have been satisfied.

The final section, part '6', is the container for all the attributes associated with this component. In principle, this section can contain name-value pairs for anything. Thus, attributes for power, area, etc. costing information could be included. In this case, one of the several 'internal' keywords has been used, namely propagation delay, which has a well defined meaning as highlighted by the waveforms.

3.1.2.2 Clocked Objects

For clocked devices, a very similar style of description is used. The main identifiable differences include the introduction of clock events and identifiable state within a timing view. In addition, a more comprehensive set of timing attributes are used. Thus, as well as the device propagation attribute, other attributes including the required setup, hold and pipeline attributes must also be expressed. The timing matrix, as with combinatorial devices, expresses the when and where of software signals mapping onto hardware ports.

The following diagram, figure 3-2, illustrates the basic understanding of setup and hold low level timing figures in relation to the propagation figure. No semantic requirement prevents the values being assigned zero or negative values, or even model devices with propagation values being less than hold. This is particularly useful since real life devices do exhibit these forms of behaviour.

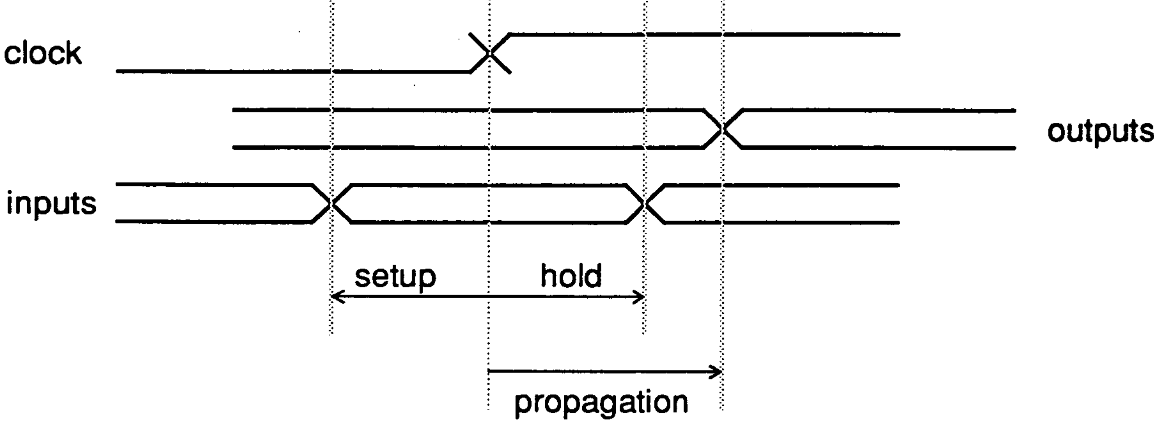


Figure 3-2. Setup, Hold and Propagation Model

The pipeline figure represents the reusability of a particular device, after the last stage of computation has occurred. The value is an offset figure and represents how many clock cycles after the last output appears, the device can be reused. Thus, in the example shown below, figure 3-3, a particular computation takes three clock cycles (from start to finish), but the device can be reused every clock cycle. The need for a pipeline figure is a direct consequence of the level of abstraction that LML has been designed for. Since the inner workings of a component are not represented by LML, it is not possible even to infer what the pipeline capability of a component is. This means that two extreme interpretations could be made. That of not being pipelined, and that of being fully pipelined. Rather than make any such assumption, an explicit figure is therefore required to represent this information. The two interpretations provide a choice of two datum-times from which the pipeline figure is able to apply. From a conceptual point of view, it is easier to start with non pipelined objects and therefore the non-pipelined datum is used to measure how soon a component can be reused.

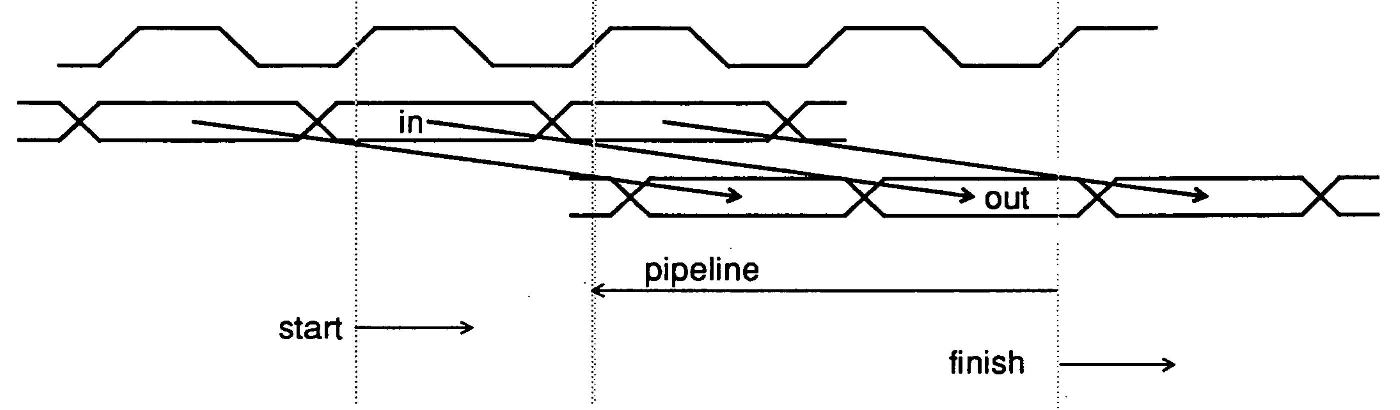


Figure 3-3. Pipeline Model

An example of a clocked LML model is shown in Figure 3-4. The graphical timing chart shows how the three software values map onto the hardware ports. The timing views are essentially a textual capture of this information.

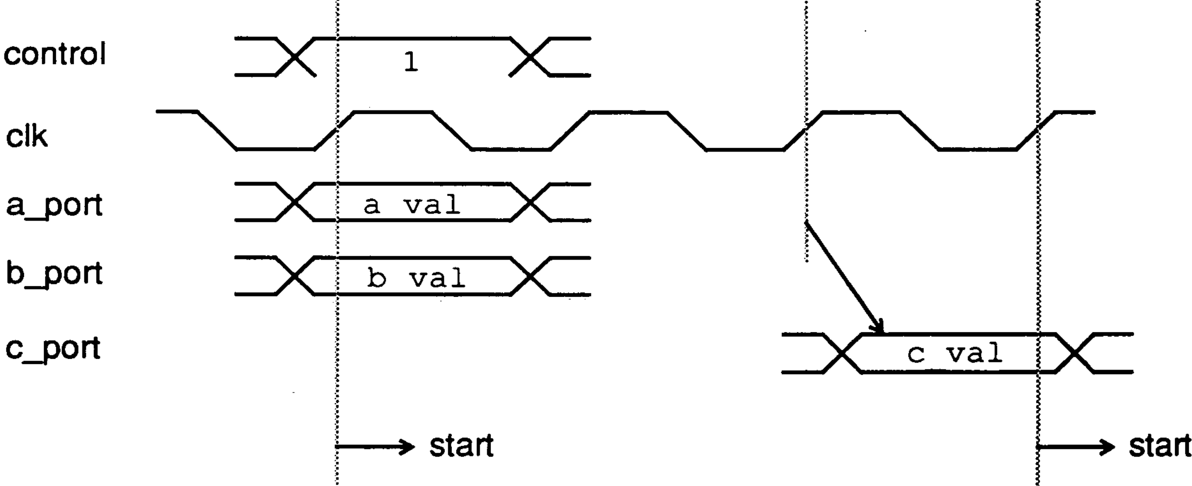


Figure 3-4. Example of Modelling a Clocked Device

Figure 3-5 shows the corresponding LML description for this timing diagram. As with the combinatorial example, this description can be used to illustrate new concepts that can be captured using LML. In part '1', the real world perspective can be seen to contain additional ports, not directly associated with any software signals within the

timing views. In this case, the additional ports are `control` and `clk`, which, as their names imply, are control signals. This model has two timing descriptions, ('2'), one for the mapping of the addition operation, and the other a subtraction operation. As can be seen in '3', each view encapsulates all the conditions that are required to execute that operation. Clock events are identified with the respective rising or falling edges, i.e., in symbolic form '/' or '\'. For every column, three sorts of element are included. First are constants, such as those on the control lines. Secondly there are variables. The clock event implies that the variables and constants must be at the required values within the constraints specified by the setup and hold attributes. The third sort of element contained within a column are generated variables, in a similar manner to combinatorial component, they are identified with the trigger '+' sign. The '.' symbolizes a wild card object, whose value is not needed to achieve the required computation. The attributes section, '4', shows three of the four timing values. The implied semantics for a required attribute, that is not given, is to default to 0.

```
component alu is
  sim_def in
    ELLA : "ALU_XXX{8}" ;
  is
    (clk : in bit, control : in bit,
     a_port : in integer, b_port : in integer, c_port : out integer)    1
  end sim_def;
begin
  time_def add is
    "$ADD"(a_val  : in integer, b_val  : in integer, c_val  : out integer)
2   control= 0,      .,       .           ;
    clk     = /,     /,       /           ;
    a_port  = a_val, .,       .           ;
    b_port  = b_val, .,       .           ;
    c_port  = .,     .,       c_val+      ;
  end add;  ______________________ 3

  time_def sub is
    "$SUB"(a_val  : in integer, b_val  : in integer, c_val  : out integer)
    control= 1,      .,       .           ;
    clk     = /,     /,       /           ;
    a_port  = a_val, .,       .           ;
    b_port  = b_val, .,       .           ;
    c_port  = .,     .,       c_val+      ;
  end sub;

  attributes_def is
    setup := 5.0;
    hold  := 2.0;                      4
    propagation := 4.0;
  end attributes_def;
end alu;
```

Figure 3-5. LML *Example of a Clocked Device*

3.1.2.3 Pipelined Objects

To illustrate more clearly pipelined objects, the behaviour of the previous example is used as the basis on which increasing pipeline capability is demonstrated as shown illustrated by the resource-time graphs in figure 3-7. Figure 3-6 shows the starting state of the design, and how it consists of two sets of two addition operations. The first two additions, #1 and #2 are related by an expression of the form (a+b)+c. The second two, #3 and #4, compute independent sums, of the form a+b and b+c. The screen snapshot shown in figure 3-6 is the result of some minor reassignment, (identified by the arrows), with resource #R1 being responsible for the additions #1 and #2, while resource #R3 handles the additions #3 and #4.

The first snapshot (a), in figure 3-7, shows resources #R1 and #R3 matched to the clocked device description as given in figure 3-5 (see section 3.3 for an explanation of how this matchmaking happens). The next two snapshots, (b) and (c) represent the same device, but with pipeline attributes of -1 and -2 respectively.

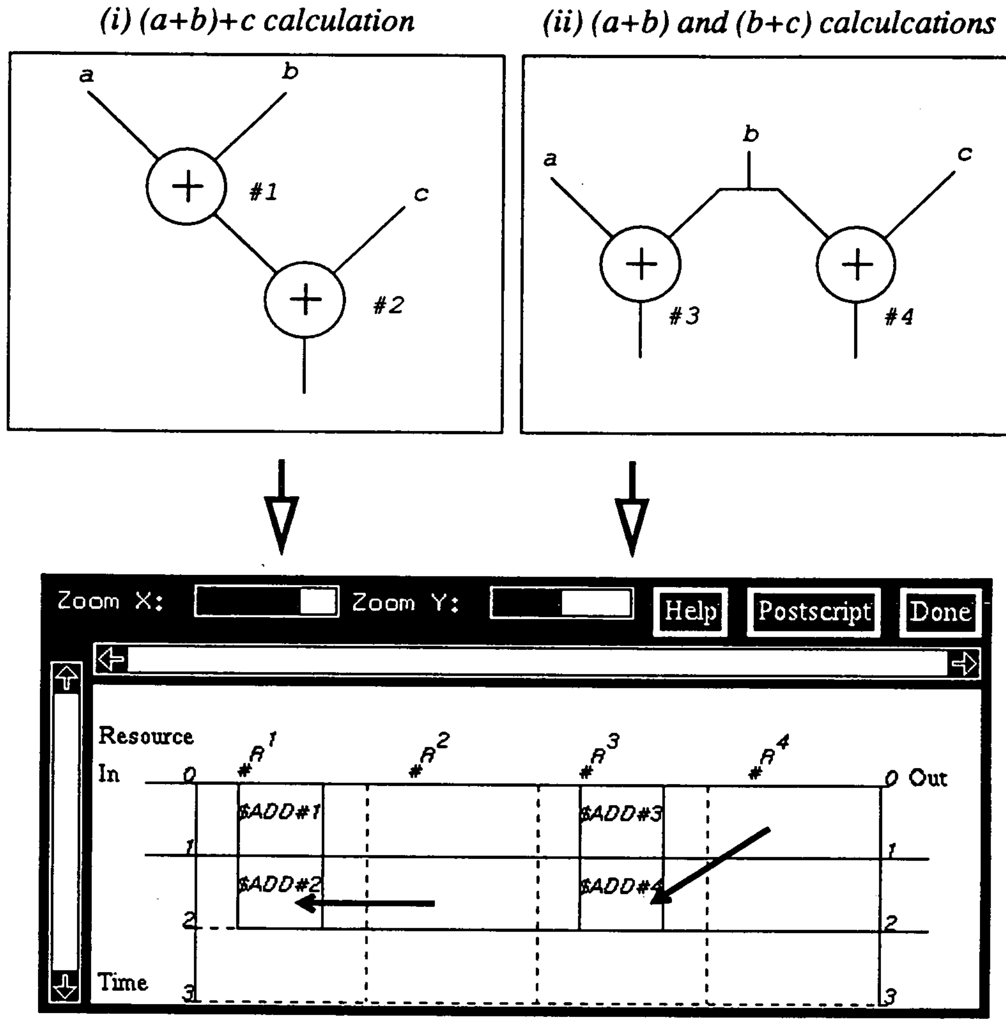


Figure 3-6. Starting State for Pipeline Example

The snapshots illustrate how the first two additions operating on resource #R1 cannot happen any faster because of the dependency in the calculation, while the second two additions operating on resource #R3 are able to overlap as the pipeline figure is modified because in this case the addition calculations are independent.

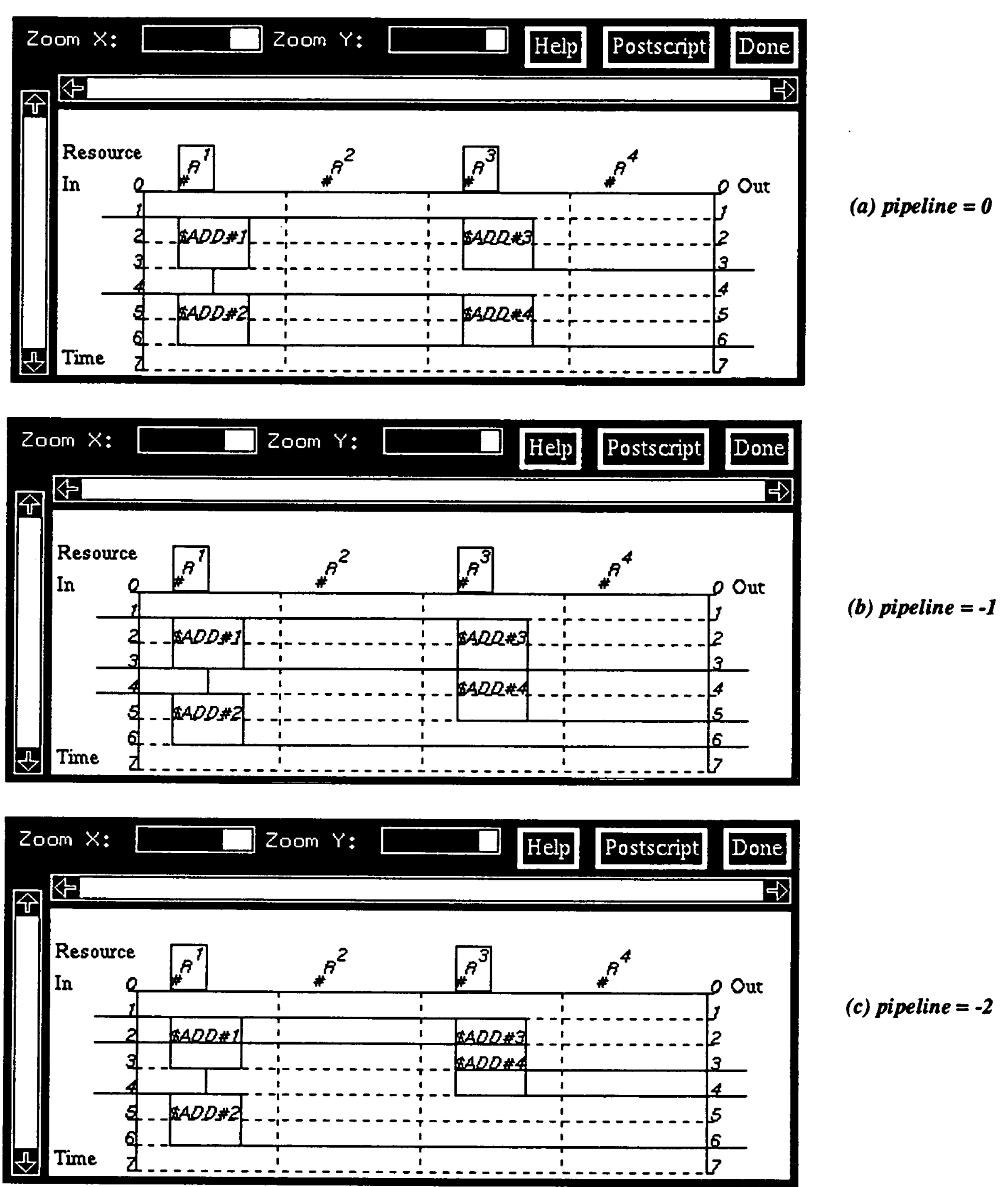


Figure 3-7. Pipeline Figure Combinations

From a graphical point of view, we can see that this information is not cleanly provided, in that the overlap causes information to be lost. As will be seen in the later sections on design visualisation, this same information can be restated graphically using parallelograms such that this information is not lost.

3.2 Structure Modelling

There are many special requirements associated with the problem of modelling structural objects, and as stated in this chapters introduction, the dominating influence of hierarchy on any design necessitates special discussion on structural modelling.

This treatment is broken down into several sections, the two most important of which being the process of creating and then generating a netlist. Important secondary issues relating to how tristates, name spaces and netlist attributes are also addressed.

What makes this discussion different is the programmatic requirements of high-level synthesis. By analysing the key features that popular HDLs provide to a user, and then analysing what a synthesis tool requires, results in a natural division of generating a netlist in two stages. Firstly, there is the process of creation and secondly the process of generating a netlist from the database created by the process of creation. The modelling process itself, through the activity of creation is relatively straightforward, but the equally important issues related to netlist generation are more involved than may appear at a first analysis.

Figure 3-8, illustrates in more detail how the creation and netlist generation activities are related, as well as the overall context within which they are used. The creation process produces a network database, from which the generation process can produce netlists in the form of schematics or the ELLA language. As well as the programming interface to drive the creation activity, other layers such as the interactive facility provide additional functionality, which is particularly useful for debugging activities. Whereas generated ELLA netlists are simply textual in nature, the figure illustrates the multitude of presentation platforms that the generated schematic could be represented in.

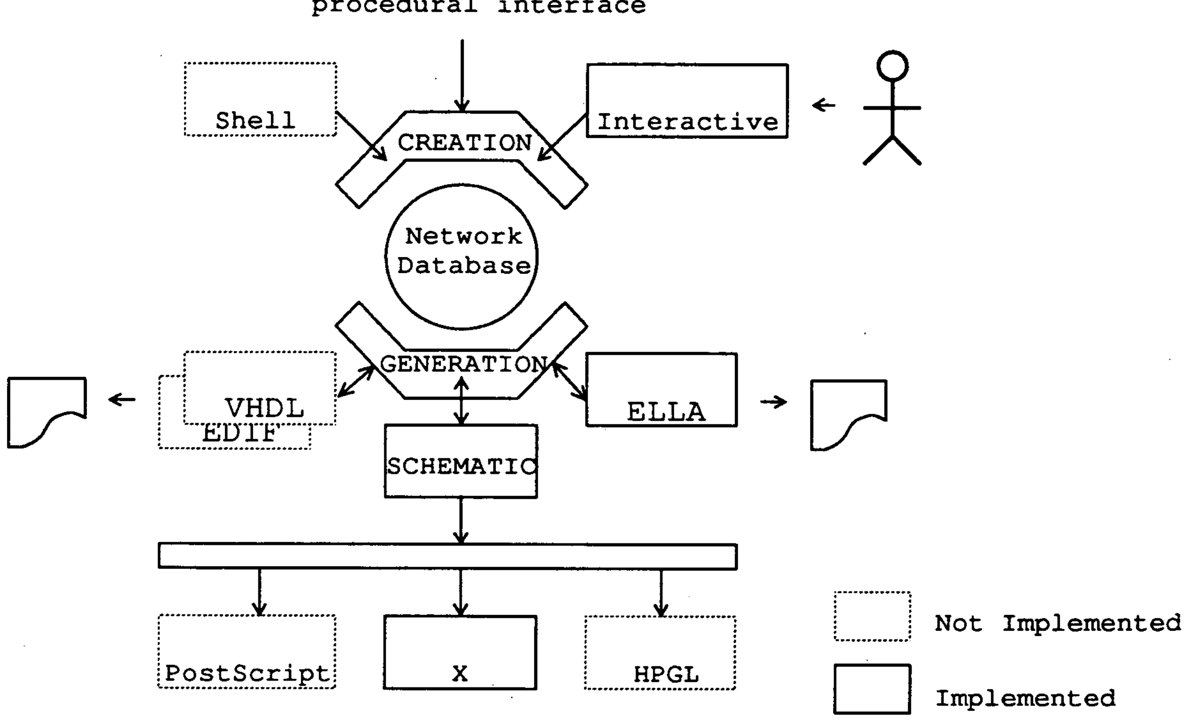


Figure 3-8. Overall Context for Creation and Netlist Generation Activities

3.2.1 Creation

All hardware description languages have a mechanism for describing structure. This usually takes the form of a textual description which must be compiled before anything useful such as simulation can be carried out with the description. These languages are targeted at human users and usually exhibit three common features. Firstly, they require that objects must be declared before their use, both in terms of general scope and visibility as well as local to the internals of a structure. Secondly a strict BNF [2] grammar must be followed, usually specified as a LALR(1) grammar. This provides a useful syntactic and semantic framework which helps for easier error detection and correction both for the compiler and the human that has to analyse the compiler output to determine where the mistakes are, and how to take corrective action. Thirdly, the names of the actual objects, such as wires and instances need careful selection to follow usually rigid name space requirements, such as ensuring no such objects take on the name of keywords. Although points one and three could be

considered to be just BNF grammar issues, note again, that these points concern the way a *user* has to relate to such HDLs with normal usage.

By looking at the requirements demanded by an automatic netlist generation tool, the reason for an explicit creation activity will become clearer. Such an interface has two primary facets. Firstly, it is procedural in a programming sense, having no need for parsing requirements. Secondly, and more importantly, the interface is directly concerned with relating created objects with each other. For example instances can be directly related with what they are an instance of, or ports can be directly related to the instance type that they are attached to. This process of creation also leads to the added benefit of being incremental in nature. This benefit cannot be understated, in that the smaller the design steps and more tightly coupled various stages in a tool pipeline, the quicker a designer can gain feedback from his actions.

A related but important point, is that unlike humans, automatic tools will tend to operate on a correct-by-construction basis. This means that strictly, the creation process has no need for checking of errors, since the tool driving the process of creation has a full understanding of the rules that need to be followed. Of course, since the tools are written by humans, there is no harm in having simple checks that will help debug such synthesis tools.

For the purposes of SAGE, the creation process was defined in terms of what were minimal requirements. An analysis of typical netlists shows that there are five types of object that need to be created. All of these are straightforward, except for the way that they must interact. The first type of object is an instance type or node, which is simply a container for the other four objects, namely instances, wires, ports and joins. Note, although SAGE did not explicitly support the notion of buses, the extension to bus objects as special instances, with wires supporting types other than simple types such as booleans or integers is in principle straightforward. The problem, with this approach is that the abstraction of a bus requires a new object being defined and an extension to the simplistic concept of wire. The following bulleted paragraphs look more closely at these five objects.

- Instance Types and Instances

What is special about instance types, is when instances local to this object need to be wired to the ports of this instance type. Three approaches could be adopted here. Special calls could handle wiring between instances and the ports on the instance type that they constitute. Secondly, all ports for all objects, including the instance type could be explicitly created, such that making any connection no longer requires special knowledge of an instance type. These first two options are unsatisfactory in creating too much information to handle. The third option is to have ports associated with an instance type, and when any connections need to be made to the instance type, it is always available as a special instance, but with a predefined name. Within SAGE, this name was called 'dot', in a similar manner to UNIX terminology, where the container directory always has an entry 'dot' to refer to the contents of the current directory. Thus, like UNIX, the driving tool must not only have an understanding of this name, it must not try to create an instance or instance type of this name. One of the classic problems that this approach overcomes, is that of specifying direct joins between input and output ports on the instance type. How this can arise is not self-evident, in that by trying to treat the ports on an instance type specially, the visibility of these ports becomes a function of how it might reasonably be used, rather than all the cases that govern how it is used. The issue of leaf level instance types is addressed later.

- Wires

As has been stated already, wires are constrained to predefined types, though in principle the set of predefined types is not limited. What is important is where information about constant signals is held. Here again, there are three choices, another special object, an instance object, or augmenting the concept of a wire with the notion of a constant value. This last choice is different from the first in that no special treatment is needed when making joins between such objects and ports. The second choice introduces an extra level of hierarchy. Thus, the last choice provides the information where needed and imposes no special treatment requirements when it comes to making connections.

- Ports

Port objects are only associated with instance types. Since leaf objects (instance types without internals) need to provide port information, the process of creating a port means a type and direction also need to be specified, Within the SAGE model, the types 'input', 'output' and 'io' have been selected as the most useful subset of all possible port directions. Languages like VHDL have a richer set of port directions, but a careful study of these shows that as well as being overly esoteric, most of the unusual ones are more related to issues of implementing a VHDL simulation system rather than being of practical use. For example, 'linkage' and 'buffer' types in the VHDL language fall into this category.

- Joins

The fifth object that can be made is a join. As its name implies, it is the relationship between a port on a given instance and a wire, in a way that models a real world electrical connection. The only difficult issue relates to the domain of joins. It would be fair to require all joins to a given wire to only connect to ports which are not connected to any other wires. Note, this definition, strange as it may seem initially, does allow multiple joins between the same (port, wire) tuple. This relates to the way data-flow can map onto structure during a synthesis tool run. For example data-flow between a parameter, say a, onto an operation op, can happen several times during an execution involving the operation op. If one resource, or hardware structure, res is allocated to execute all the invocations of op, then a synthesis tool can be seen to request a hardware path between the source of a, and the consumer op for as many times that there are invocations of op. Though simplistic, this example illustrates how multiple identical joins can legitimately occur.

If the restriction on the domain of joins is dropped, the gain in indirectly supporting the concept of wire aliases outweighs the security that a restricted domain provides. Again, an example from a synthesis tool driving such an interface highlights its usefulness. Taking the same example as before, but this time the second invocation of op has a different parameter, say b. Thus the suppliers of a and b will both have associated wires that will become aliases, when a connection is made to the associated input on the resource res.

There are several additional subtleties that need to be explored before this methodology towards a creation interface can be considered to be complete. The first concerns scope. This model supports only two levels of name space - global instance types and the names used within each instance. This is no way near as flexible as scope rules provided by languages such as ELLA and VHDL, but for an automatic interface it provides an adequate degree of flexibility. Within this global name space exist leaf level libraries. An empty instance type does not automatically mean it is a library object, since the incremental framework can allow that instance type to be revisited to define its contents. The method used to tag instance types as library cells happens during the instance type creation process. In languages like VERILOG [36], it is possible to navigate around an instance hierarchy within the declaration region of what is the equivalent of an instance type. Dubious as the benefit of such a facility, such probing facilities are not supported, instead the equivalent effect is achieved by passing the wires that need to be monitored through the instance hierarchy via ports. The final point that can be made concerns the error processing that incremental creation can require. Since the types and direction of ports are specified, checks that are simple to state such as multiple drivers on a wire could be carried out. Unfortunately even this check can become computationally expensive in that a full elaboration of a circuit must be made in order to correctly determine the number of drivers on a given wire. Thus any checks that are not O(1), are considered too computationally expensive.

More for information than further discussion, figure 3-9 shows the actual creation procedural interface. It shows the several commands that are needed to create the network database by manipulating the five objects defined earlier.

```
with TEXT_IO, NETWORK; use TEXT_IO, NETWORK;
package CREATION is

  DEBUG_CREATION : BOOLEAN := FALSE;

  -- in all routines that take STRINGS, the following must be true, they must:
  -- (1) have no leading, trailing or internal space characters
  -- (2) have STRING'FIRST = 1


  procedure START_CREATION(NAME : STRING; LEAF : BOOLEAN := FALSE);
  -- if building an object that needs to be identified as leaf, then
  -- LEAF should be set to TRUE. Only MAKE_PORT makes sense for a leaf object.
  procedure END_CREATION;
  -- these two are  repeatedly called, to build objects
  -- START_CREATION can be called again without having to call END_CREATION


  -- if there is no valid creation activity, the following 'active' commands will not work

  procedure MAKE_WIRE(WNAME, WTYPE : STRING);
  -- WTYPE can be:
  -- (1) "bit", "int" or "flt" or
  -- (2) as above followed by & ";" & constant, eg. bit;1 or flt;0.7
  -- (3) null, in which case it defaults to "bit"


  procedure MAKE_INSTANCE(INAME, ITYPE : STRING);
  -- (1) /XXX/YYY/ZZZ  => it is comp out of the library, which is instanced.
  --                      (this is as the name in the "LIBRARY_COMPONENT_FIELD" of babble)
  -- (2) XXX           => then it is a local comp type which is instanced.
  --                      (exceptions will probably be made for MUXES, TRIS etc.)
```

```
procedure MAKE_PORT(PNAME, PDIR, PTYPE : STRING);
-- this defines the ports for the io block name of the current 'START_CREATION' node
-- this is called successively to name all the ports
-- PDIR is "in", "out" or "io"; if "" then it is assumed to be "io"
-- PTYPE is "bit", "int" or "flt"; if "" or ANYTHING else then it is assumed to be "bit"


procedure MAKE_JOIN(WNAME, INAME : STRING; PORT : POSITIVE);
procedure MAKE_JOIN(WNAME, INAME : STRING; PORT : STRING);
procedure MAKE_JOIN(WNAME_1, WNAME_2 : STRING);
-- note:
-- (1) the types of the WNAME and port on INAME must be the same
-- (2) if the INAME is '.', then it is referring to the iomake (as defined by MAKE_PORT)
-- (3) making a join to a port that is already joined, or making a join between two
--     wires, is effectively equivilant to creating an alias


-- this is a private dump facility for checking purposes
procedure LIST_DESIGN(NAME : STRING; FILE : in FILE_TYPE := STANDARD_OUTPUT);
procedure LIST_DESIGN(FILE : in FILE_TYPE :=  STANDARD_OUTPUT);

...

end CREATION;
```

Figure 3-9. Creation Procedural Interface

3.2.2 Attributing for Netlist Generation

As shown earlier in figure 3-8 on page 37, the creation process has a second and equally important interface, that of actually generating a netlist. As with the requirements that drive the creation process, the requirements here are also very focused. With an object centred model of a structure, there are requirements to traverse the structure and annotate the database of objects. Whereas the former is a common and expected requirement, the second, is conceptually unusual, in that the normal behaviour is to replicate the entire database with this additional information. By looking at the disadvantages of not having this approach, the significant extra benefits of having an annotateable database can be seen.

To help this discussion, a simplistic netlist generation algorithm needs to be developed, that requires new names conforming to the rules defined by the target language to be generated for all objects. How these names are created is not important, (though issues concerning name spaces are discussed in greater detail later in section 3.2.3.1).

The netlist generation algorithm can be viewed as two passes. The first pass of the algorithm must generate new names for all the nodes. The second pass involves analysing the instances. From the point of view of the instance, two node types are of interest - that which it is contained within, and the type of the instance. For the purposes of this algorithm, only the latter is of interest. The instances node name again needs to be referred to but, depending on where the new node name information was stored, can cause the traversing algorithm to exhibit O(1) to O(n) behaviour - where n is the number of nodes. In normal netlisters, this would be O(n), where an encountered instance would require a full traversal of the database representing the information associated with nodes - in this example, the information being the new node names. If, on the first traversal, information could be directly attributed to the node objects, then when traversing the instances, the information will be available in O(1) time. This example has only focused on node names, but the same arguments follow for the interrelationships between the remaining objects of instances, ports, wires and joins.

This attributing facility relies on two general methods. Firstly, full viewing facilities and secondly indirection routines. The viewing routines provide a statement of the connectivity and information associated with the network database. The commands fall into two general categories, those that extract information and those that summarise information. Examples of the former include getting the name of an object, while for the latter a query of the form 'is there any instance or instances inside this node' is a typical example.

The real cornerstone to the attributing method is provided by indirection routines. This mechanism lets netlist extraction software fully integrate with the creation database by providing data hooks on which netlist extraction software can on the fly create and maintain local state information. Figure 3-10 (a), outlines how conventional extraction packages operate - basically on a view only basis. With indirection fields available, as indicated in (b), the reverse arrows (flowing left to right), illustrate information attached to the creation database, and how it relates back to the netlister specified data structures, if any.

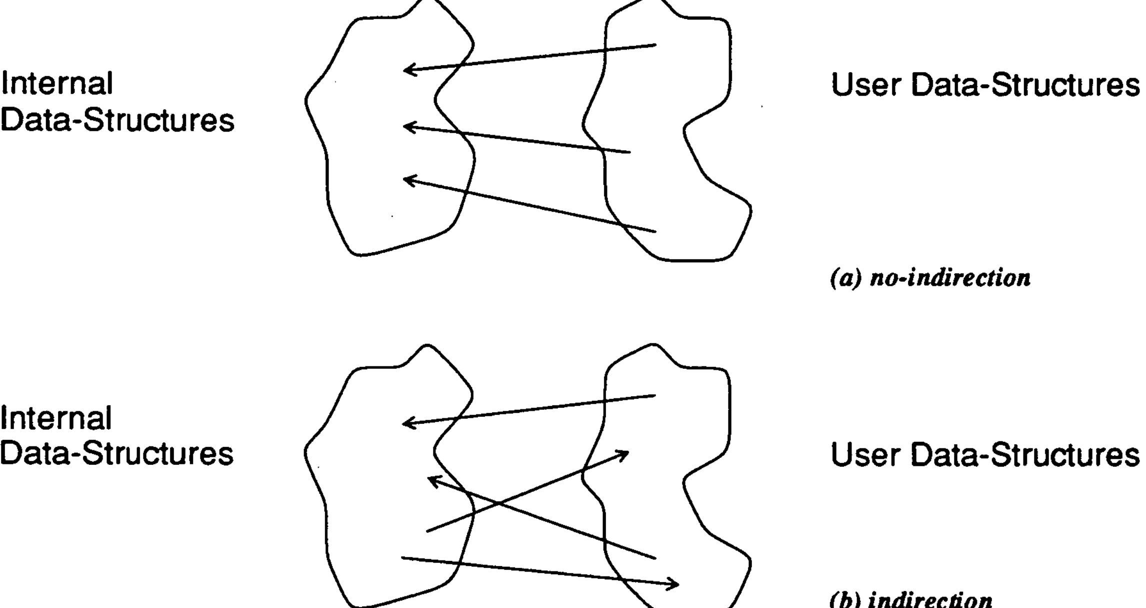


Figure 3-10. Indirection and No-Indirection Comparison

In practical terms, in order to make use of these hooks, it is necessary to supply all the netlister specified data structures as well as reference objects to them. The following

code illustrates the from of the ADA generic that can be used to achieve this indirection.

```
generic

  type USER_NODE_T is limited private;
  type USER_NODE_PTR_T is access USER_NODE_T;
  ...

package HOOKS is

  procedure PUT(
    THIS : USER_NODE_PTR_T;
    IN_THIS : NETWORK.NODE_PTR_T
  );

  function GET(
    FROM_THIS : NETWORK.NODE_PTR_T
  ) return USER_NODE_PTR_T;

  ...

end HOOKS;
```

Figure 3-11. Generic Outline for Indirection Support

Note, that two routines per object are supplied as a result of any instance of this generic, the total being ten, since there are five basic objects. The key point here, is that since the creation database has no prior understanding of the netlister data-structures, it is necessary to use normally unsafe programming constructs such as unchecked conversion, but by wrapping an extra level of procedural functionality the requirements of being strongly typed and therefore safe are reacquired. On a smaller, but equally important point, without a scheme of this form, the addition of third party netlister software would require recompilation of a large part of the creation software and relinking of the top level object binary. A more detailed exploration of how indirection is used, is described in section 6.5 on page 171, since the generality of the approach has also been used in managing the graphics databases used to achieve design visualisation.

3.2.3 Generation

Given an internal database, an analysis of the target languages drives the algorithms that must traverse this database in order to produce correct translated code. In order to

ensure a broad discussion that relates to real world languages, three of the most popular HDLs have been considered, namely VHDL, ELLA and EDIF. The broad scope of these languages helps highlight all the key translation requirements. Note though, since schematic generation is generally unconstrained being an end in its own right, it does not impose the same requirements for accuracy as generated netlists which must work with minimal interference in the target simulation environment to be useful.

ELLA, being functional in nature, has several styles of netlist description. If there is no feedback within a group of instances, it is possible to state the structure much like an equation. This form means that the names of the instances are no longer required. The more practical style is the process of declaring all the objects such as instances and wires before they are used in join constructs. Since ELLA is functional, the process of making wires can be difficult, since they must correspond to input ports or outputs of already declared instances. That is, no notion of a placeholder to be expanded later on in a description. This same functional style also means bidirectionals rely on being a mathematically pure, but practically cumbersome notion of functions sets and function types. Another point to note about ELLA, is its restrictions on identifiers. Instances must start with an uppercase letter, while wires must start with a lowercase letter. By convention the whole of the identifiers follow the case of the first letter. Within a given scope, identifiers must be no longer than 255 characters, but be unique within the first 20. Even manually handling such requirements can tend to be a very error prone activity.

The next language to be considered, VHDL is much more flexible in the restrictions placed on the identifiers. As well as being case insensitive, no restriction on size is placed, though actual implementations do usually place such a limit around the 256 character mark. Unlike ELLA, VHDL does support the concept of an explicit signal, which means creating a basic netlist is far easier since no sorting between inputs and outputs is needed as in ELLA. With the much neater concept of a resolution function, bidirectionals need no special treatment. Where VHDL is distinctly different from ELLA, is the requirement for configuration statements, which defines a path through the instantiation hierarchy. This is more useful for a human designer developing

separate architectures for a given entity, for selecting say a behavioural level architecture on one occasion, and a second register transfer level architecture on another occasion.

EDIF having been designed with interchange in mind, is the most flexible of the three languages being examined. Storing its information in views, it has an explicit netlist view which, like the creation network database, stores all the information in a fully cross-correlated form, albeit textually. This cross-correlation happens in that signals have information about which instances they are attached to, rather than this information being only implied by the description. As far as the language construction goes, identifiers can be of any length, but only the first 255 characters are significant. If the first character of an identifier is not a letter, it must be preceded by '&'. Since EDIF is LISP [10] in style, normal implementations are not prone to the maximum line limit restrictions that ELLA and VHDL implementations generally have. This is a practical point and relates to how the various languages lend themselves to being compiled. ELLA and VHDL, with real users in mind, are structured so that lines have to be a sensible length for a user to be able to sensibly read, edit, print etc. such files, and this usually leads to the compiler writers placing a hard limit as to how many characters a line may contain. It should be noted that this limit is wholly arbitrary, since languages like ELLA and VHDL have grammars defined that strictly place no limit on line length. Where the LISP style helps significantly over ELLA and VHDL, is that there are no reserved words, (except within a 'keyword name def' construct).

Probably the best example illustrating the power of the creation abstraction, is how it can be used for supporting not only language based netlist generation, but also schematic generation. The problem at a simple level, is association of shape and location to instances, location to ports and a path defined as a series of connected segments to a wire. Creating a picture database is only half the story. The second half concerns when this picture database is interrogated. Unlike the behaviour of the netlisters for HDLs, this action is incremental in nature and requires on the fly probing of the creation database. This also works two ways, in that an object within the creation database could be modified, and that change needs to be passed to the picture database structures.

One of the common themes highlighted by looking at the three languages, is the specialist requirements on name space handling. The first of the next three sections looks at all the issues of mapping names in detail, and shows the development of a general approach to mapping names that can apply beyond just the netlist generation problem domain, but to many other languages. The next section looks at the specific issues involved in translating from this creation database to ELLA, since it represents the hardest of the three cases. Following this is a closer look at how schematic generation happens within this creation framework.

3.2.3.1 Name Space Problems

Languages like VERILOG actually support an explicit escape mechanism for identifiers in foreign languages. The reason why this is not normally done can be seen from the way the compilation technology works. Normal compilers have two basic parts, a token digester and a set of grammar rules. It is at the token digesting stage where an attempt is made to minimize the number of context sensitive dependencies, and this usually equates to just strings, comments and two character tokens such as '!=' (not equal). The common feature of these context sensitive fields is the need for special mark characters that start and finish the token and that differentiate it from other context sensitive tokens. For example, in VERILOG, the start character is '\' and the terminating character is any whitespace character. The decision not to have this feature in all languages can therefore be seen to be a simple pragmatic decision, because its an irritating thing for a user to have to do for every single identifier !

Without such a mechanism in the target language, some form of name mapping is required. If one considers two domains, a source domain and target domain, then rules need to be specified about each domain, that governs how names from the source domain are mapped to the target domain. Clearly, a simplistic approach of having a target domain that simply maps all source names to the lowest common denominator, such as a letter followed by a sequenced number would solve the problem. In fact many translators adopt this approach, but it is usually based on the premise that human users will have no interaction with the resulting netlist. As a premise, it is wholly inadequate, since the premise fails to note that when there are problems, the

mangled names in machine generated output will have to be examined to determine the cause of the problem, and the cost in time and resources can become very significant. This same cost can effectively frighten potential users away from using advanced CAD tools. This is because the time and effort lost when encountering a problem would appear to be out of all proportion to the benefit of the tool - whether the problem is a fault of the user or the software being run. It is from this observation, that the requirement that as much information as practically possible should be maintained arises.

To solve this mapping problem, the several common elements illustrated in the range of languages for which output is desired, were identified and captured in a general purpose ADA abstraction. Several parameters are provided, which specify the nature of the target language as well as certain aspects of the source language. Once instantiated, the naming software provides several routines which effect the mapping and allow interrogation of the database of old and new names. Figure 3-12 shows an example of an instantiation used by the ELLA mapping software.

```
package NAMES is new NAMING(
  ALPHABET_SET_T => ALPHABET_T,
  FIRST_LETTER =>
     "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz",
  REMAINING_LETTERS =>
     "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789",
  SEPARATER_LETTERS => "_",
  UNIQUE_MAKING => "0123456789",
  LENGTH_LIMIT => 255,
  UNIQUE_LIMIT => 20,
  SOURCE_CASE_SENSITIVITY => FALSE,
  TARGET_CASE_SENSITIVITY => TRUE
);
```

Figure 3-12. ELLA ***names package***

A number of points can be noted from this generic package instantiation. Names are considered to be of the lexical form:

```
first_letter { [separater_letter] remaining_letter}
```

In a generated name, a separator is never repeated adjacent to itself. This occurs even if the target language is able to support repeated separator characters.

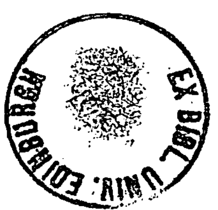
Unique making letters are used if a conflict occurs, and are usually placed at the end of a name, and preceded with a unique making character. These unique making letters follow a counting scheme, of which two styles can be identified. There is number counting, as the example would produce, with postfixes that would be of the form:

`_1, _2 ..., _9, _10, ... .`

The second style, is that of letter counting, where there is no 'zero' element equivalent. To have this style of counting, the first element of `UNIQUE_MAKING_LETTERS` should be repeated. For example having `UNIQUE_MAKING_LETTERS => "aabcdef"`, would generate postfixes of the form:

`_a, _b, ..., _f, _aa, _ab, ... .`

As well as being able to define source and target case sensitivity, the mapping process can be directed to do useful things with the original case. Four case directives are available: `MIXED`, `UNMIXED`, `UPPER` and `LOWER`. `UNMIXED` uses the case of the first letter encountered to determine the case preferred. The application of these options, can become complex, and in practice is an iterative activity. Figure 3-13 illustrates the mapping of two names, which each require unique mappings in the presence of given values for the target case sensitivity and case directive facilities. The reason why source case sensitivity does not apply at this stage is because when a name is being mapped, it is blindly assumed that it is already unique compared with previous source names encountered. Clearly, this would not be the situation in this example if source case sensitivity were false. One of the reasons for this behaviour is so that unnamed objects, like wires in a netlist, can all be given the same name, say `wire`, and for each such object, a mapping is requested. This would produce a series of new names like `wire`, `wire_1`, `wire_2`, `wire_3`, and so on.


EX BIBL. UNIV. EDINBURGH

Target Case Sensitivity:	False		True	
	Mixed	Unmixed	Mixed	Unmixed
Words that have been mapped:	`SaGe`	`SAGE`	`SaGe`	`SAGE`
`SaGe`	`sAgE_1`	`sage_1`	`sAgE`	`sage`
`sAgE`	Upper	Lower	Upper	Lower
	`SAGE`	`sage`	`SAGE`	`sage`
	`SAGE_1`	`sage_1`	`SAGE_1`	`sage_1`

Figure 3-13. Name Mapping Example, Using Target Case Sensitivity and Case Directives

The database used to create unique names, is in fact the second of three conceptual databases, and is called 'naming'. Figure 3-14, illustrates the three databases, and how they are related. There is a domain containing all the 'old' names that require mapping, and there is a destination 'new' database which contains the results of a mapping. 'Naming' represents the various stages an old name might go through, before a unique name is found and passed to the 'new' database. The source case sensitivity directive is used by a 'map old to new' request, by checking to see if it already exists in the 'old' database and returning the mapping if found, else generating a new unique name. In the example shown in figure 3-13, if source case sensitivity was false, and both identifiers had a 'map new to old' process applied, then the second identifier ('`sAgE`'), would in all cases map to the same identifier as that produced for '`SaGe`'. The implication is that when source case sensitivity is false, a case folded version of the identifier is stored in the 'old' database, (but not in the 'naming' database).

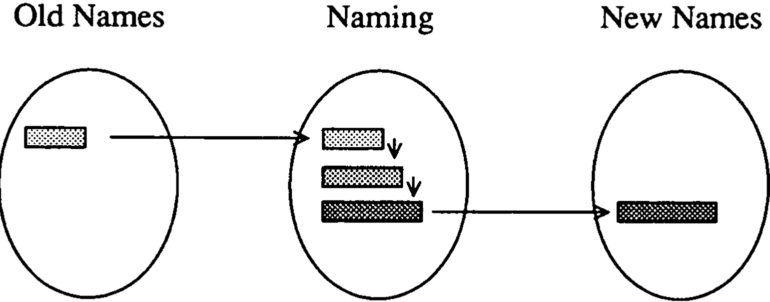


Figure 3-14. The 3 Databases Used in Unique Name Generation

Returning to figure 3-12 of the generic instantiation, there is a clear distinction between how many letters the target system will accept, and of those, how many are considered unique. If the UNIQUE_LIMIT is not specified, then it defaults to the LENGTH_LIMIT, as one would expect.

Several procedures and functions are made available to the user of the generic. The following is a list of these routines, with the name of the function or procedure giving some idea of what is achieved.

```
function GET_UNIQUE_NAME(
     START : STRING_T;
     CASE_IS : CASE_T := DEFAULT_CASE; ...
) return STRING;

function MAP_NEW_TO_OLD(NAME : STRING_T; ...) return STRING;
function MAP_OLD_TO_NEW(
     NAME : STRING_T;
     CASE_IS : CASE_T := DEFAULT_CASE; ...
) return STRING;

function EXISTS_OLD(NAME : STRING_T; ...) return BOOLEAN;
function EXISTS_NEW(NAME : STRING_T; ... return BOOLEAN;

procedure REMOVE_OLD(NAME : STRING_T; ...);
procedure REMOVE_NEW(NAME : STRING_T; ...);
```

Figure 3-15. Functions and Procedures present in Names Generic

In order to give a clearer idea of how and what the mapping can achieve, the following is some example output for the given package NAMES, as shown in figure 3-16, but with two changes. The changes set LENGTH_LIMIT to 10 and UNIQUE_LIMIT to 5. This helps highlight the handling of the extreme or edge effect cases much better. Some of the examples, also show the mapping happening with case directives present.

```
             # 'ALONG'        ->   'ALONG' #
           # '__apple'        ->   'APPLE' #
           # 'ap__ple'        ->   'AP_PLE' #
                 # '__'       ->   'A_1' #
               # '/__/'       ->   'A_2' #
             # 'hello'        ->   'HELLO' #
             # 'hello'        ->   'HEL_1LO' #
            # 'OrAnGe'        ->   'ORANGE' #
            # 'OrAnGe'        ->   'ORA_1NGE' #
            # 'OrAnGe'        ->   'OrAnGe' #
          # '__pear__'        ->   'PEAR' #
# 'alongwirenameagain'        ->   'alo_1ngwir' #
             # 'along'        ->   'alo_2ng' #
             # 'alo_2'        ->   'alo_3_2' #
             # 'ALONG'        ->   'alo_4ng' #
     # 'alongwirename'        ->   'alongwiren' #
            # 'OrAnGe'        ->   'orange' #
```

Figure 3-16. Examples of Unique Name Generation

For a given target netlist, the general algorithm is to have a names database (i.e. all 3 core databases) for each level of scope. For a given node, the starting state of the 'naming' database will consist of all language keywords - (b) in Figure 3-17. Not taking such keywords into account is probably one of the most common cause of problems with tools that translate from one HDL to another. ELLA has over 40 such keywords, and these with the 41 'local' definitions for such things as leaf cell primitives and types are effectively marked as being reserved. Note, this means that the 'old' and 'new' databases are therefore empty. The next stage is to map all the node names, to ensure that there are no name clashes across the hierarchy of a design, to a copy of this keywords names database, producing result (c). Although it may appear this step is superfluous, without it, no integrity of instance type names across a generated netlist would be possible. The resulting two databases, (b) and (c), are combined creating a new keywords database and copying in the contents of the 'new' node names database. The resulting database, (d), is used for mapping the names of wires and components for each node in a network database. As seen earlier, names which are template in nature - i.e. those objects that have no assigned name, are added directly to the naming database, resulting in the generation of the next name in sequence.

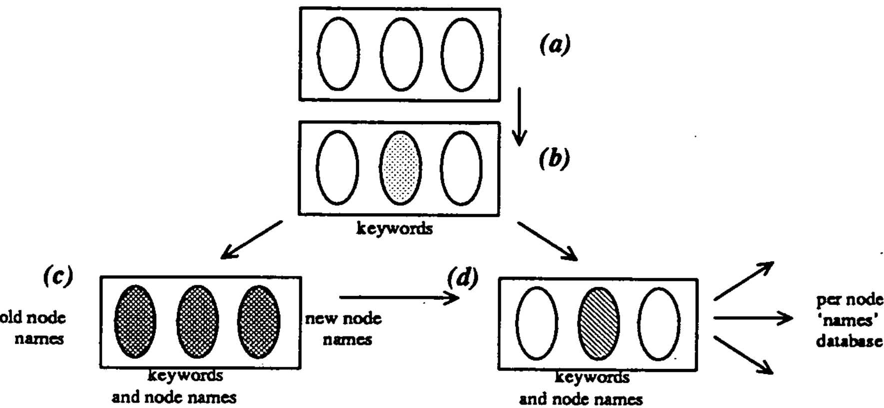


Figure 3-17. Names Database Management

3.2.3.2 Generating ELLA

In order to generate correct ELLA from a given creation database, there are several technical problems that have to be addressed. The emphasis on generating correct code, is very important, since it is straight forward to produce something that looks like ELLA, but would cause a compiler to produce many errors. In fact, compared with VHDL and EDIF, ELLA causes more problems in translating from the creation abstraction because of its slightly more rigid and formal structure.

The following bulleted points look at these problems, and assume an understanding of ELLA. These problems are:

• ELLA being functional in style, requires inputs to be sorted from outputs. Although it appears straightforward, this means creation objects with no inputs and/or outputs need to be handled carefully, since all ELLA code must have some input and output always.

• ELLA has no pre-defined types or library parts. Therefore representative objects have to be created.

• The concept of bidirectionals and tristates is not the normal one used by designers in real life. Of all the problems to resolve, this is the most difficult, because the constructions in ELLA are not user friendly, and have an impact in nearly every stage of the resulting ELLA.

• ELLA is fussy about unconnected inputs - which usually arises from the netlist being generated before all the synthesis stages have occurred. (This problem also arises if there are problems with the synthesis routines, and not all inputs are connected). Consequently, although it would be fair to expect all inputs should be wired up, because of the process of synthesis, unconnected inputs can be expected. In a similar way, outputs that have a fanout of 0 can be assumed to arise for similar reasons. Thus, some means of sweeping up unused inputs and outputs and categorising them as unused (as say) controller ports is needed. For any given node being mapped, the number of such ports will be unknown and thus a generic ELLA block is needed to sweep these signals up. (Particularly in the case of unused outputs, without any such facility to sweep up these signals, there would be no way to do a first level check by compiling the resulting ELLA).

• There is the philosophical issue of whether the generated output should be human readable. That is, well organised to the extent that its meaning is clear - a feature which is very useful for debugging the output. This implies care and attention to details like spacing, addition of signpost comments and how the ELLA comma separator and full stop terminator characters are used. This last point refers to the two general styles of marking declarations as being distinct - namely a token that acts purely as a separator, as in 'a; b', or that acts as a terminator, as in 'a; b;'. ELLA, unlike VHDL, uses the former extensively, but as can be seen, the decision as to when and where to place a separator requires foresight of the fact that there is another item to display.

• Regardless of whether the netlist output is neat or not, the order of the constructs used must be carefully defined since ELLA requires all objects to be defined before they can be used. This requirement would appear to be reasonable, but ELLA provides no way to declare wire objects. Instead wires are implemented as a node aliasing concept (using LET), and the names given to instances are also treated as node aliases. The solution turns out to be in broad outline straightforward, but at the expense of having to choose the lowest common denominator style of ELLA coding namely a MAKE/LET/JOIN style.

All the above problems have been addressed in the resulting ELLA generator, as well as the usual name space problems.

The ELLA types defined are shown in figure 3-18. The type 'dummy' handles functions that have no inputs and/or outputs. The remaining three categories shown, map one for one with the three core types used within SAGE. The important aspect to each of these three, is the development of a function type (FN TYPE), based on a bus object that can take on one of three states. As with all bus objects which will in general have multiple drivers feeding it (i.e. fanin > 1), at least three states can be defined. The bus can be undriven, or, if there is only one driver active on the bus, then the bus can be said to be driven with an associated valid bus value (the '&' symbol in ELLA indicates this association). The third state a bus can be in is that of an unknown value, where there are two or more drivers trying to determine the state of the bus. Note, although this bus concept could be developed further to reflect reality more closely, this would be counterproductive since it provides a framework within which errors in the results of synthesis can easily be found in a simulation. A good example is that reality could support two drivers on a bus with the same value, but the results of synthesis are generally designed to ensure that there is only one active driver on a bus at all times. The function type object is described later.

```
TYPE
  dummy = NEW (void).

TYPE
  bit = NEW (f | t),
  bit_bus = NEW (bit_undriven | bit_driven & bit | bit_unknown),
  bit_conn = bit_bus -> bit_bus.

TYPE
  int = NEW i/(-32768..32767),
  int_bus = NEW (int_undriven | int_driven & int | int_unknown),
  int_conn = int_bus -> int_bus.

TYPE
  f_man = NEW f_m/(01000000..10000000),
  f_exp = NEW f_e/(-128..127),
  f_sign = bit,
  flt = (f_sign,f_man,f_sign),
  flt_bus = NEW (flt_undriven | flt_driven & flt | flt_unknown),
  flt_conn = flt_bus -> flt_bus.
```

Figure 3-18. ELLA ***Types***

There are three predefined types of object that synthesis generally requires. These are tristate buffers, 2:1 multiplexers and memory. Figure 3-19 shows the form of the ELLA code for what these primitives look like for signals of type bit.

```
FN BIT_BUFFER = (bit: in, bit: control, bit_conn: out) -> dummy:
  BEGIN
    JOIN
      CASE control OF
        t : bit_driven&in,
        f : bit_undriven
      ESAC -> out.
    OUTPUT (void)
  END
.

FN BIT_MUX = (bit: a, bit: b, bit: control) -> bit:
  CASE control OF
    t: a,
    f: b
  ESAC
.

FN BIT_RAM = (bit: in, bit: load) -> bit:
  BEGIN
    FN REGISTER = (bit) -> bit: DELAY(?bit,1).
    MAKE REGISTER: register.
    JOIN
      CASE load OF
        t : in,
        f : register
      ESAC -> register.
    OUTPUT register
  END
.
```

Figure 3-19. Netlist Generation Primitives: Buffer, Mux and Ram

Looking at the model for the BIT_BUFFER, apart from the odd looking construction of a dummy output, note how the tristate output port is viewed from the function type perspective, rather than the equally valid bus associated type. In short, this choice allows bidirectionals to be treated in a systematic fashion. Figure 3-20 shows the two possibilities, the second of which is the one used.

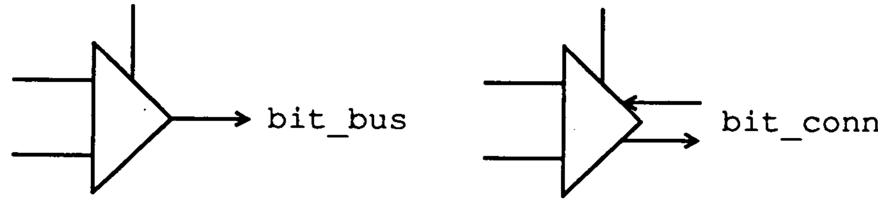


Figure 3-20. Different ELLA *Views of Tri-state Buffers*

Buses are defined as objects to which multiple drivers may be attached. In ELLA terms, buses are function types which could appear in the input or output area of a function definition. The choice is arbitrary (reflecting the clumsiness of the concept). The ELLA netlister uses the inputs field. Given a bus, there is the usual need for a bus resolution algorithm. The ELLA text in figure 3-21 encapsulates the core of the bus resolution algorithm, namely how 2 drivers are handled on a bit_bus.

```
FN BIT_RESOLVE = ([2]bit_bus: in) -> bit_bus:
  CASE in OF
    (bit_undriven, bit_driven&bit) : in[2],
    (bit_driven&bit, bit_undriven) : in[1],
    (bit_undriven, bit_undriven) : bit_undriven
  ELSE
    bit_unknown
  ESAC
.
```

Figure 3-21. ELLA *Bit Resolution Algorithm*

In order to use this resolution unit on a bus with n connections, a MACRO, as shown in figure 3-22 is required to define a function set (FNSET), which can replicate and wire up the required number of resolve blocks. Note, how this method ensures proper resolution of buses that can traverse a complex hierarchy.

```
MAC BIT_BUS{INT n} = FNSET [n] ((bit_bus: in) -> bit_bus):
  IF n = 1
  THEN
    in
  ELSE
    BEGIN
      MAKE [n-1]BIT_RESOLVE: block.
      JOIN (in[1],in[2]) -> block[1].
      FOR INT i=2..n-1
        JOIN (block[i-1],in[i+1]) -> block[i].
      OUTPUT [n](block[n-1])
    END
  FI
.
```

Figure 3-22. The FNSET bit_bus Macro

In total, twelve distinct ways to wire up an ELLA description can be identified. These are illustrated in figure 3-23, separated into two categories: connections to inputs, including bidirectional points, and connections to outputs. 'u.c' represents an unconnected port, while 'const' represents a constant value. The ELLA '//' and 'IO' terminology might appear confusing because they are infrequently used. '//' represents the extraction of the value part from an associated type. 'IO' represents the notation needed to say pass the argument object as a bidirectional, rather than the default behaviour, of only the output part of a bidirectional. The two marked as 'not really possible', are a reflection of the fact if they are present, the bus has a well defined value and by implication must have no other drivers.

Connections To Inputs and IOs				**ELLA Representation**
(1)	IO	⇒	I	`// ripper`
(2)	IO	⇒	IO	`IObus[]`
(3)	I	⇒	I	`name`
()	I	⇒	IO	`driven&... (not (really) possible)`
(4)	u.c	⇒	I	`controller line name`
(5)	u.c	⇒	IO	`IO XXX_UNDRIVEN block`
(6)	const	⇒	I	`(...value...)`
()	const	⇒	IO	`driven&(...value...) (not (really) possible)`

Connections To Outputs				ELLA Representation
(1)	IO	⇒	O	`// ripper`
(2)	I	⇒	O	`name`
(3)	u.c.	⇒	O	`controller line name`
(4)	const	⇒	O	`(...value...)`

Figure 3-23. Connection Types

There is no need to make a distinction between ports on instances and those on the defining node, except in the case of IO to IO. Figure 3-24 illustrates the difference. The main point is that a different style of ELLA code is required, compared with that for an IO connection associated with an instance.

The bus object represented in these pictures also highlights the way all connected signals that are associated with a bidirectional, are automatically collapsed onto the same bus object. This is because the bus function set is responsible for determining the final state of the bus, and so it must be able to 'see' all connections, particularly through the instantiation hierarchy. It is for this point that the style of ELLA is different, since IO connections on a node must be made so that the ELLA simulator is able to traverse through the instantiation hierarchy and accurately determine the final value of a bus.

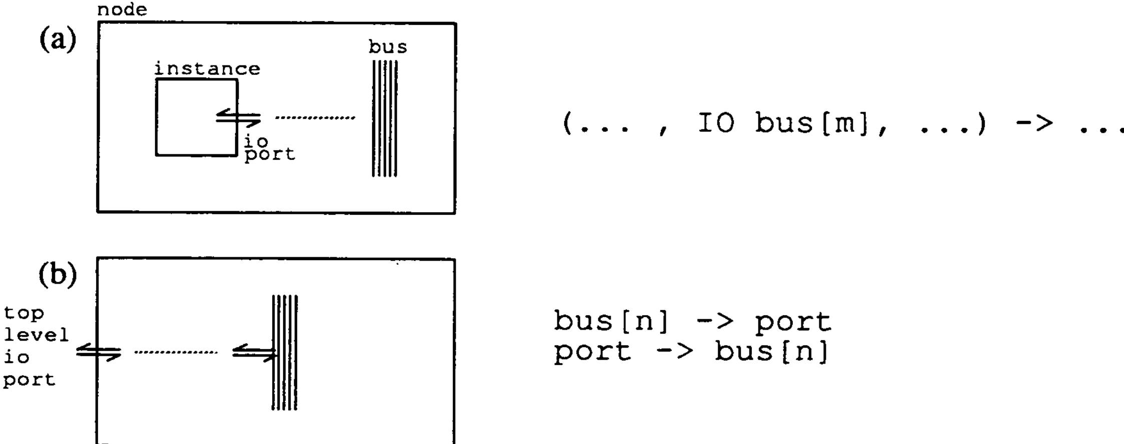


Figure 3-24. IO -> IO Connections at Node and Instance Levels

For all unused inputs and outputs, a block of the form shown in figure 3-25, will be produced in the generated ELLA code.

```
...
  #** controller hook **#
  MAKE
    CONTROLLER{
      dummy, 0, dummy, 0, flt, 2,
      dummy, 0, dummy, 0, flt, 1
    }: controller.
...
```

Figure 3-25. Unused Inputs, Outputs and IOs Consumer

This represents dummy code, for which manually written code has to be generated to complete the full functionality of the node that it forms part of. As mentioned earlier, for the ability to be able to at least compile the generated ELLA, a generic block needs to be defined. The code for this block is shown in figure 3-26, and illustrates the fact that if the unused inputs and outputs are intentional, the block faithfully reproduces this behaviour by feeding out ELLA anonymous values.

```
MAC CONTROLLER
  {
    TYPE ti1, INT ii1, TYPE ti2, INT ii2, TYPE ti3, INT ii3,
    TYPE to1, INT io1, TYPE to2, INT io2, TYPE to3, INT io3
  } =
    (
      [IF ii1 = 0 THEN 1 ELSE ii1 FI]ti1: i1,
      [IF ii2 = 0 THEN 1 ELSE ii2 FI]ti2: i2,
      [IF ii3 = 0 THEN 1 ELSE ii3 FI]ti3: i3
    ) ->
      (
        [IF io1 = 0 THEN 1 ELSE io1 FI]to1,
        [IF io2 = 0 THEN 1 ELSE io2 FI]to2,
        [IF io3 = 0 THEN 1 ELSE io3 FI]to3
      ):
(
  IF io1=0 THEN [1]?to1 ELSE [io1]?to1 FI,
  IF io2=0 THEN [1]?to2 ELSE [io2]?to2 FI,
  IF io3=0 THEN [1]?to3 ELSE [io3]?to3 FI
).
```

Figure 3-26. ELLA ***Controller MACRO***

The generic form of an ELLA netlist is shown below. It represents all the components that can be present in non-leaf generated ELLA. Obviously, depending on the nature of the network database, certain sections will not be present in the final output.

```
GEN_FN_HEADER;
GEN_INTERNAL_MAPPINGS;
GEN_CONTROLLER;
GEN_BUSSES;
GEN_INSTANCES;
GEN_CONTROLLER_OUTPUTS;
GEN_BUS_DRIVER_VALUES;
GEN_SINGLE_DRIVER_LETS;
GEN_CONTROLLER_INPUTS;
GEN_BUS_TO_IO_BLOCK_JOINS;
GEN_HARD_BUS_DRIVER_JOINS;
GEN_INSTANCE_JOINS;
GEN_OUTPUT_STATEMENT;
```

Figure 3-27. Generated ELLA ***Netlist Structure***

```
FN LATCH = (bit: sbar, bit: rbar) -> (bit, bit):
BEGIN
  #** identifier mappings **#
    COM[[
      fn name was: 'latch'
      'a' -> 'a'
      'b' -> 'b'
      'q' -> 'q'
      'q' -> 'q_1'
      'qbar' -> 'qbar'
      'qbar' -> 'qbar_1'
      'r' -> 'r'
      'rbar' -> 'rbar'
      's' -> 's'
      'sbar' -> 'sbar'
    ]]MOC
  #** no controller hook needed **#
  #** no buses **#
  #** user instances **#
  MAKE
    NAND: a b.
  #** single driver lets **#
  LET
    s = sbar,
    r = rbar,
    q_1 = a,
    qbar_1 = b.
  #** component wiring **#
  JOIN
    (s, qbar_1) -> a,
    (q_1, r) -> b.
  #** ELLA block output statement **#
  OUTPUT
    (q_1, qbar_1)
END.
```

Figure 3-28. Untouched Automatically Generated ELLA ***Example***

To help illustrate some of the layout style, as well as the connectivity methods, figure 3-28 has the full untouched ELLA code generated for a small example, consisting simply of a latch. The example was generated using the interactive interface using textual commands like 'mp' for make port, as shown in the flow diagram in figure 3-8 on page 37.

All objects that have been matched, or that are know internally to SAGE, have ELLA comment lines generated. Figure 3-29 shows an example of such output that is generated, and effectively corresponds to the output generated for leaf level objects.

```
...
# * /user/sage/library/models/aluf.lib, used in makes as ALUF_XXX_8 #
#   io spec: (bit: clk, bit: control, flt: left, flt: right) -> flt: #

# * $BUF, used in makes as BUF #
#   io spec: (int: a, bit: ctrl, int_conn: q) -> dummy: #

# * $MUX, used in makes as MUX #
#   io spec: (int: a, int: b, bit: ctrl) -> int: #

# * $RAM, used in makes as RAM_1 #
#   io spec: (int: right, bit: load) -> int: #
...
```

Figure 3-29. Example of Leaf Level Mappings in Generated ELLA

As a consequence, it is necessary to set up suitable ELLA 'import' and 'export' directives to map the relevant commented line to the required primitive.

3.2.3.3 Mapping Schematics

The issues involved in mapping schematics from the network database are much easier. This is because both network and schematic databases have the same style of representation. There are two places where there is slight difficulty. Firstly, in the case of wires, where there is usually not a one to one mapping, because a wire in the schematic is composed of multiple horizontal and vertical segments. Secondly, as with ELLA, the ports need to be sorted so that all inputs appear on the left, outputs on the right and bidirectionals on the bottom edge of a component. (The situation is flipped left-right for the defining node). Further issues related to generating

schematics are discussed next chapter, in section 4.8.4 on page 114, in much greater detail.

3.3 Matchmaking and Unification

During the various stages of synthesis, there comes a point where the leaf level objects need to be replaced by real hardware. This process is termed matchmaking. When the library elements are structural in nature, then a more complex form of matching termed unification can occur.

3.3.1 Match Searching

In simple terms, a given function is searched for in a given set of libraries. The location for the libraries are determined by an external `UNIX` environmental variable called, `SAGE_LIB_PATH` that is simply a list of the directories containing library components. For a given call, if a function is found, then the correct number and type of ports are also checked. The mechanism for checking if the behaviours are the same, is a simple case-insensitive check of the two names. The corresponding component of the behaviour is returned for every match found. This resulting list, if any, is presented to a user in a panel. With very rich libraries, it can be imagined how this list is ordered by analysis of the attributes associated with each of the behaviours in the returned list. An example of the list returned by matchmaking is shown in figure 3-30. It represents the sort of panel that might be produced when requesting a match for one of the adders selected in the pipeline example shown earlier in figure 3-7 on page 35.

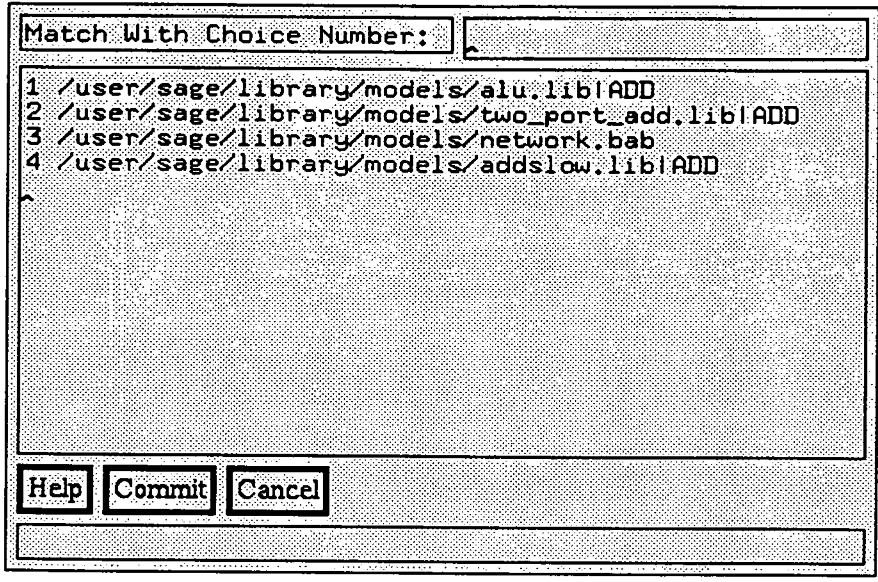


Figure 3-30. Matchmaking Selection Panel

Note the entries in this panel. Objects 1, 2, and 4 represent LML objects, which have placed after the vertical bar, the name of the associated timing view - in this case being ADD. Clearly, 2 and 4 are adders, but 1 is a multi-function unit. Entry number 3 represents a network object. Since the object being matched to in this example is leaf in nature, the unification approach is not needed to determine equivalence.

This matchmaking approach can be compared with that found in modern synthesis tools as examined in Chapter 2, where there is a complex matching against combinatorial expressions. This arises because these synthesis tools usually restate all but the memory elements of a system in terms of boolean equations. The SAGE approach is different primarily because of the much higher level objects that it has been designed to handle. Nevertheless, there is huge difference in the fidelity of these two styles of matchmaking, which suggests that the method used in SAGE is inadequate.

3.3.2 Unification

The behavioural hierarchy will not generally match the structural hierarchy that is finally produced. Therefore, in a general sense, parts of the hierarchy are removed and

then reconstructed in a different way. One of the ways to rebuild hierarchy is to use unification on structure. This method is also useful for matchmaking based on some common element in a behaviour, such as a butterfly operation in an FFT design.

To give a flavour of the problems involved in this style of matching, the task of matching two similar designs was examined. The only difference being the names associated with the nodes and wires, and their order. The process of unification is then one of finding the corresponding names between the two designs for the wire and instance names. With PROLOG [1], the problem can be remarkably succinctly captured as illustrated in figure 3-31.

```
weak_islist([]).
weak_islist([_|_]).

append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1, L2, L3).
permute([],[]).
permute(L,[H|T]) :-
  append(V,[H|U],L),
  append(V,U,W),
  permute(W,T).

unifyi(A,B,P) :-
  unifyi(A,P),
  permute(B,P).
unifyi(inst(_,F,_,_),inst(_,F,_,_)).
unifyi([],[]).
unifyi([A|B],[C|D]) :-
  unifyi(A,C),
  unifyi(B,D),
  not(weak_islist(A)),
  not(weak_islist(C)).

unifyw(_,[],_,[]).
unifyw(A,[B|C],D,[E|F]) :-
  ((A=B,D=E);(A\=B, D\=E)),
  unifyw(A,C,D,F).

unifywa([],_,[],_).
unifywa([A|B],C,[D|E],F) :-
  unifyw(A,C,D,F),
  unifywa(B,C,E,F).

convolve(_,[],_,[]).
convolve(A,[B|C],D,[E|F]) :-
  unifywa(A,B,D,E),
  convolve(A,C,D,F).

convolve_wires([],[]).
```

```
convolve_wires([A|B],[C|D]) :-
  convolve(A,[A|B],C,[C|D]),
  convolve_wires(B,D).

get_wires([],[]).
get_wires([inst(_,_,W1,W2)|T],COMBINE) :-
  get_wires(T,TAIL), append([W1,W2],TAIL,COMBINE).

unifyf(A, B, INS1, NEWINS2) :-
  device(A,I1,O1,INS1),
  device(B,I2,O2,INS2),
  unifyi(INS1,INS2,NEWINS2),
  get_wires(INS1,W1),
  get_wires(NEWINS2,W2),
  append(W1,[I1,O1],WIRES1),
  append(W2,[I2,O2],WIRES2),
  convolve_wires(WIRES1,WIRES2).
```

Figure 3-31. Unification Program

The top level rule, `unifyf`, takes two arguments `A` and `B` that represent two aribitrary netlists and then tries to confirm if the two netlists are equivalent by producing the corresponding netlist elements in the `INS1` and `NEWINS2` variables. The rule first tries to find corresponding instances of the same type using `unifi`, and then for this given combination of instances confirms if both instance lists, `INS1` and `NEWINS2`, have the same point to point wiring by using the rule `convolve_wires`.

The representation of the two netlists are simple `PROLOG` rules. This is illustrated in figure 3-32 with the code for two `DTYPE`'s, `dtype1` and `dtype2`. By comparing `dtype1` with the schematic, the format of the example can be easily seen. The second dtype, `dtype2`, has different instance and wire names, as well as the order of the instances placed in a 'random' order to better illustrate the way the unification process must happen.

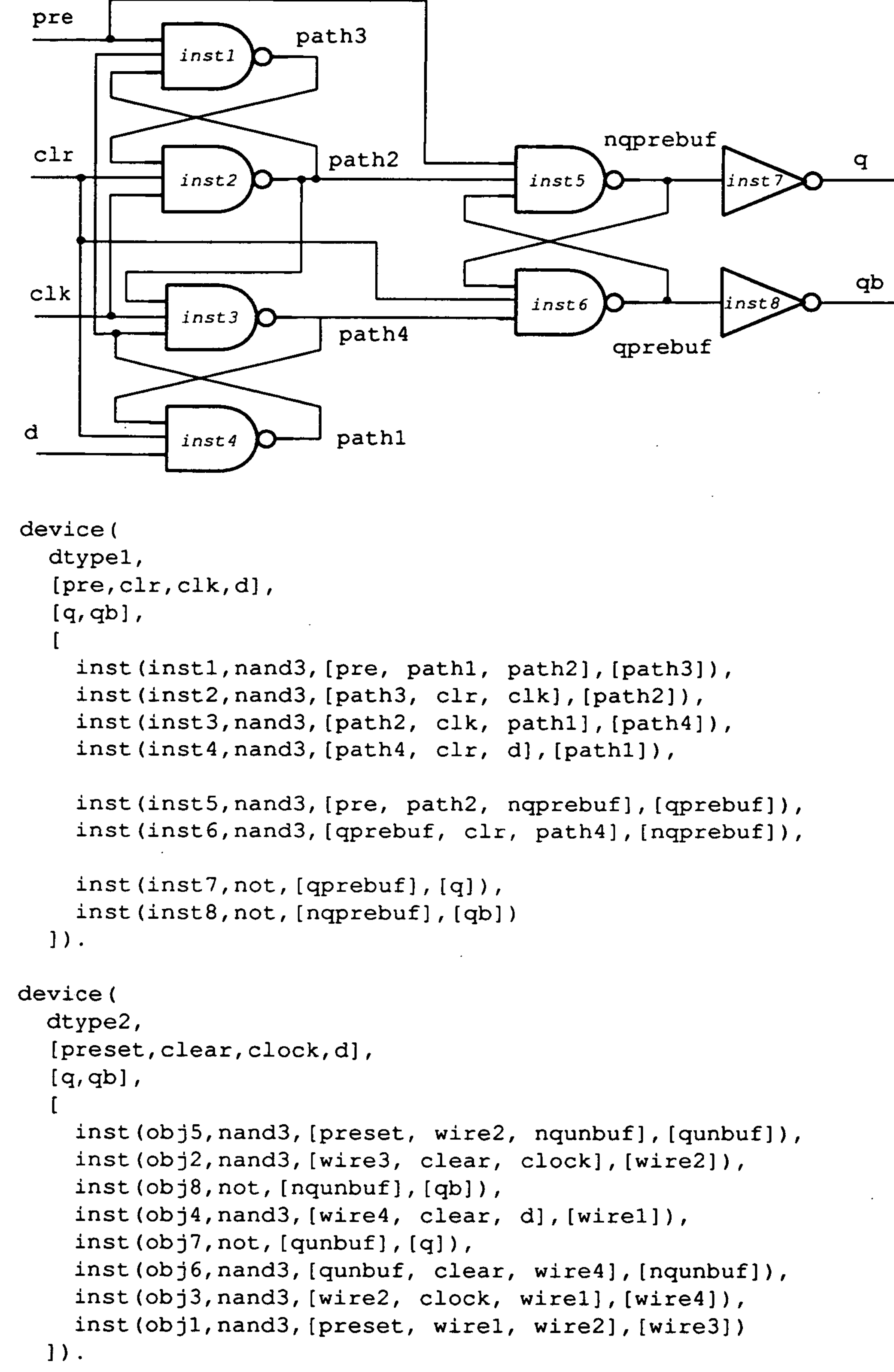


Figure 3-32. Unification Example

In this example there can be only one match as illustrated by the PROLOG run session shown in figure 3-33. Although the instance names were different between the two dtype rules, the postfix numbers assigned were the same, a fact which is reflected in the run session by the one to one matching of the instance sequence numbers.

```
SB-Prolog Version 3.1
| ?- ['source.p'].

| ?- unifyf(dtype1, dtype2, A, B).

A = [\
inst(inst1,nand3,[pre,path1,path2],[path3]),\
inst(inst2,nand3,[path3,clr,clk],[path2]),\
inst(inst3,nand3,[path2,clk,path1],[path4]),\
inst(inst4,nand3,[path4,clr,d],[path1]),\
inst(inst5,nand3,[pre,path2,nqprebuf],[qprebuf]),\
inst(inst6,nand3,[qprebuf,clr,path4],[nqprebuf]),\
inst(inst7,not,[qprebuf],[q]),\
inst(inst8,not,[nqprebuf],[qb])]\
B = [\
inst(obj1,nand3,[preset,wire1,wire2],[wire3]),\
inst(obj2,nand3,[wire3,clear,clock],[wire2]),\
inst(obj3,nand3,[wire2,clock,wire1],[wire4]),\
inst(obj4,nand3,[wire4,clear,d],[wire1]),\
inst(obj5,nand3,[preset,wire2,nqunbuf],[qunbuf]),\
inst(obj6,nand3,[qunbuf,clear,wire4],[nqunbuf]),\
inst(obj7,not,[qunbuf],[q]),\
inst(obj8,not,[nqunbuf],[qb])];
no
| ?-
```

Figure 3-33. Run of Unification Example

Ken Thompson has an automobile which he helped design. Unlike most automobiles, it has neither speedometer, nor gas gage, nor any of the numerous idiot lights which plague the modern driver. Rather, if the driver makes any mistake, a giant "?" lights up in the center of the dashboard. "The experienced driver", he says, "will usually know what's wrong."

4. Design Visualisation

The third major interface aspect of high-level synthesis, next to library modelling and netlist generation, is that of design visualisation. In its broadest description it is the mechanism of distilling or condensing design information into a form that a designer can find useful. A very important part of this is interaction, where a user can walk around a very tight feedback-modify loop.

In recent years a huge amount of visualisation work in many disciplines other than just the field of synthesis has been carried out. Rather like many other concepts of recent years, visualisation has many of the connotations of a buzzword in vogue rather than being strictly a new discipline. In a similar manner to so called new technologies like client-server, object orientated design and virtual reality, visualisation has been present in many disciplines for many years. But only recently, with the general availability of more powerful computers has this discipline been identified by its presence in many areas of technological development. From the basic adage that a picture saves a thousand words, disciplines as far apart as radiography, climatology to zoological animal tracking now employ common visualisation techniques for presenting information in an easily assimilated form. In the field of high level synthesis, the use of visualisation has been driven by the increasing complexity of designs. Large amounts of information need to be presented and managed in a way that achieves design objectives.

This chapter has been broken down into several parts. The first section deals with the broad concepts related to visualisation from a historical perspective. Section 4.2 focuses on what visualisation must provide to a user. Section 4.3 looks at the SAGE data model to give an idea of what some of the visuals presented in later sections are representing. Section 4.4 looks at visual representation from a generic point of view.

Since one of the most complex aspects is actually drawing graphs, section 4.5 looks at this problem. Issues of inter visual interaction and dynamic visualisation are addressed by the following two sections. Section 4.8 then focuses on details of the visuals as addressed in the SAGE system. The last section of this chapter, section 4.9, addresses the issues of how a user interacts with the information presented in the visuals.

4.1 Development of Visualisation

As has been mentioned in chapter 2, a vast amount of work in the areas of chip layout and schematic capture have become the traditional visualisation methods in the field of electronic system design. More recently, there has been a swing back to textual entry, but supported by language sensitive editors. This move reflects the power of languages, but also the lack of technology in visualising the complexity of the concepts provided by such new languages. This lack of technology manifests itself in two ways. Firstly, the mapping of language constructs to suitable graphical objects, and secondly, the software technology needed to support the actual mapping in an efficient and effective manner. The point about efficiency is important, since simply manually drawing out a graphical interpretation can be considered visualising, but with a turn-around measured in hours, such visualisation then becomes only useful for demonstrating concepts in places like theses !

It is from the many disciplines that make use of visualisation concepts, that the commonality in many of the presentation ideas and concepts has fostered what is currently perceived as a new discipline. In fact, considerable progress has been made by the support of numerous general purpose tools. From tools that support graphical pipelines with a rich set of image operations like AVS, to more general purpose tools such as spreadsheets with a rich assortment of bar, line and pie charting techniques, a rich assortment of visualisation tools have been developed. But, being embryonic in nature, such tools fail to support easily the wide demands of visualisation for high-level synthesis, principally because such tools have been focused on requirements that are application domain specific. Thus what the following sections will address is the development of the design concepts that need to be presented and why specifically for

high-level synthesis requirements. Chapter 5 and 6 will partly focus on the mechanism needed to support such graphical concepts.

There are many factors that control the scope of visualisation techniques. The most dominating influence is that of hardware. Only in recent years has the de-facto engineering workstation been identified as a high-resolution bit mapped screen, with mouse and keyboard support. Prior to this, the normal interface was that of a character based screen or terminal with keyboard. The natural extrapolation from this is towards stereoscopic interfaces with even larger display surfaces with extensive use of sound and spatial awareness. Of the several human interfaces, silly as it may currently seem, this still leaves scope for using smell, taste, position and voice for even more esoteric interfaces. In step with the actual interfaces, has been an explosion in computing performance.

Even during the development of the work in this thesis, mid-range computing power for SIMD machines has grown by about a factor of a 100. The benefits of this computing power is two-fold. Not only is interactive performance increased, but activities that were considered as batch can now be migrated into the design flow, such that the stages it forms a link between can now be seamlessley integrated. This second advantage is subtle, in that it is not simply a case of the software working faster, but a recognition that a certain stage in the software now works so fast, that slight modifications to the software will lead to greater throughput. The tools that rely on 'what if' analysis, are the best examples of where this extra benefit arises.

Next to computer hardware development, the second most dominating influence on general visualisation performance, is that of software. Careful use and application of algorithms can easily beat even the mammoth increases in computing performance. The now classic example of this, can be seen by comparing UNIX with Microsoft Windows, which on a like for like hardware basis seems to go 10 to 100 times slower. In the same realm, compilers in the PC world that do not even have the luxury of virtual address spaces always significantly outpace their UNIX conterparts, in not only being faster to compile but producing higher quality machine code that is generally smaller and faster than those from UNIX machines. Thus, careful

development of algorithms is essential and has therefore become a backbone to the work presented in this thesis.

There are many psychological factors that drive the form of an interface. In terms of straight information assimilation, the quicker the step from understanding information to an action will result in a corresponding greater gain in productivity, because the decay time of information held in the brain has less and less effect. For example, a 10% improvement in speed, will give a greater than 10% improvement in productivity. Other psychological factors relating more to the form and function of the interface tools are aspects that can only be measured over time, and, as was the case with SAGE, has an impact by feedback on the tool implementation.

Given these two major constraints, namely hardware performance and software quality, (with the second generally governed by time and effort that is needed), this chapter focuses on *realisable* visualisation techniques for high-level synthesis. Figure 4-1 illustrates the perceived productivity benefit to a designer. Whereas lower specification hardware will fail to provide a productive environment, simply because of lack of performance, higher performance machines fail because of their novelty and hence lack of software support. With the passage of time and improved software, the peak of productivity moves higher and migrates to higher performance hardware. It is interesting to note how lower performance hardware can become even less productive because some of the improvements will rely on having high-performance hardware. This region between the two dotted lines, represents realisable visualisation techniques explored in this chapter. Another interesting point that this figure highlights, is the reason for the reluctance of designers to adopt new design approaches. By not changing tools, a designer still gains productivity through improvement by better software quality and improved hardware, and therefore unless a new tool gives significant extra advantage there is little reason to adopt a new tool.

(In some ways this is the story of SAGE: it's clever, but the extra benefit is just not enough).

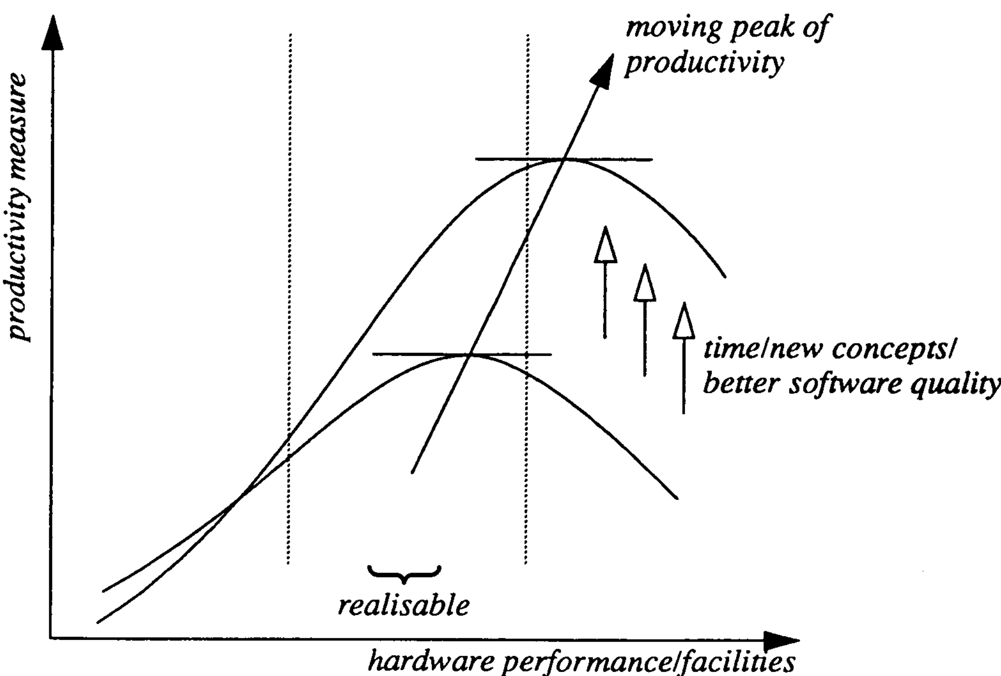


Figure 4-1. Productivity vs Software vs Hardware

One of the added bonuses during the duration of this thesis, has been the adoption of software standards by hardware manufactures. In particular, the `X Window System` [55] and `UNIX` (in the form of `POSIX`) are now industry standards. Nevertheless the development of the concepts have been directed by the performance and facilities provided by mid-range workstations. Thus the extra benefit that a `CRAY` supercomputer, for example, could provide has not been explored. In short the work presented is for designers in the main, rather than the currently rarer designer who has access to supercomputers.

4.2 Visuals Overview

Of the whole user interface, two broad divisions can be made, as illustrated in Figure 4-2. There is that which focuses on what a designer observes, through design visualisation, referred to as visuals, and then the process of taking a designers input through the `UIMS` (User Interface Management Services). The two categories are not completely disjoint, but have significantly different behaviours to justify separate

treatment. Additionally, the field of UIMS has generally matured to the extent that it does not pose significant technical problems, especially through the use of off the shelf toolkits like Motif [57] and OpenLook [51]. Chapter 5 looks at some of the remaining problem issues with the UIMS more closely.

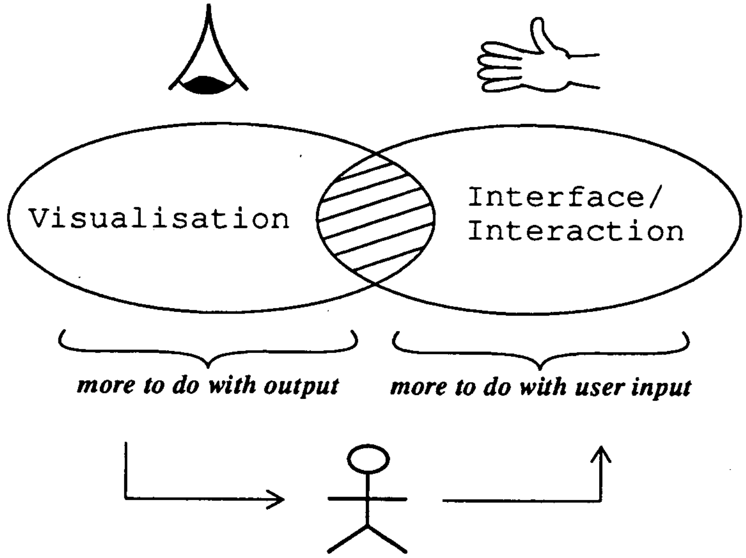


Figure 4-2. Design Loop

The main vehicle through which design visualisation happens is that of a visual. In crude terms it corresponds to a 2 dimensional area capable of conveying design information. In more precise terms, there are five key features of visuals, most of them obvious, but nevertheless worthwhile enumerating.

• Firstly, a visual shows the current state of a design. This does not preclude historical information since it contributes to the current state, and might well be required when backtracking through design steps.

• Secondly, it carries information in a human understandable form. The emphasis here is how easily it is possible for this information to be comprehended by a designer.

• Thirdly it is amenable to interaction at various levels. This interaction refers to incremental synthesis actions that directly impact the design database.

• Fourthly, aspects of the visual can be modified to highlight different aspects. The key point here is that the substantive part of the design is not modified.

• The final feature of a visual is that it is a springboard from which design decisions can be made.

Of all these uses, the last is the most important, since there is minimal point in providing visual information that does not help in taking the next design step. Unfortunately, this is the most subjective, and it is only after extensive usage that the full usefulness of a visual can be ascertained. A good comparison here is with computer languages, where only when one is experienced with a language, can a fair statement about its effectiveness be made. Thus many C [8] programmers never have a kind word to say about ADA [3], even though they have never used the language.

What the following sections present are the key visuals that were identified for the SAGE model. In particular, the visuals have been identified to be particular cases of superclasses that capture general concepts, and a full exploration of these general ideas is presented. As well as what the visuals are and their theoretical basis in terms of their superclasses, the following sections also focus on the intra and inter-visual issues that arise. On the intra-visual issues, this involves the information that can be displayed and how it is modified and explored. The inter issues concern how the visuals can be related with each other.

4.3 SAGE Data Model

In order to place correctly in context the developed visuals, a clear understanding of the key SAGE concepts is needed. This section briefly explains several of the key SAGE concepts.

The SAGE model is dominated by the need to show a designer the 'where and when' of operation (or more formally, call) activity. With these quantities, a designer has key information from which to make improvements to the space consumption and also the time consumption by respecifying another place and/or time for an operation to happen. This activity happens in a resource-time-like framework. Clearly, such a re-specification can only happen if no resource clashes happen, or any data-flow clashes.

The SAGE model has several key concepts and entities, at the heart of which is the notion of a 'design' unit. A design unit is a container object that has two basic parts. Firstly it can capture the control flow and data flow intentions of a designer in one or more behaviour units. Secondly it is a container for how it is implemented in a structural unit - i.e. the resource requirements. The behavioural side separates the data flow from the control flow, by containing the data flow in blocks. More precisely, these blocks correspond to *basic blocks* as defined formally in compiler technology terms [2]. Like an ALU design can have multiple behaviours (add, subtract etc.), there can be multiple behaviours associated with each design (i.e. in a similar manner to different behaviours in LML). Each basic block contains a data flow graph of function calls connected by data arcs. The structural side contains a netlist of components connected by connections.

Function calls are calls to behaviours in other designs, which is the mechanism used to support hierarchy in designs. In tandem with this hierarchy, is that of structural hierarchy since components are instantiations of structural objects. Thus, in the same way as a component has a definition as represented by its structure, a call has a definition as represented by its behaviour. Many function calls can be instantiations of the same behaviour, and many components can be instantiations of the same structure.

The process of design is that of exploring the possible mappings between the given behaviour of a design, to its structural implementation and then selecting one of these choices. In broad terms, this is achieved by automatic or user directed creation of components and connections that achieve the requirements of the function calls. This process means that at the end of the design stage, all calls are bound to particular components. Figure 4-3 illustrates these primary SAGE data model objects and their

interrelationships. Using the approach of correctness by construction, all modifications to a design unit must be fully self-consistent before they are allowed.

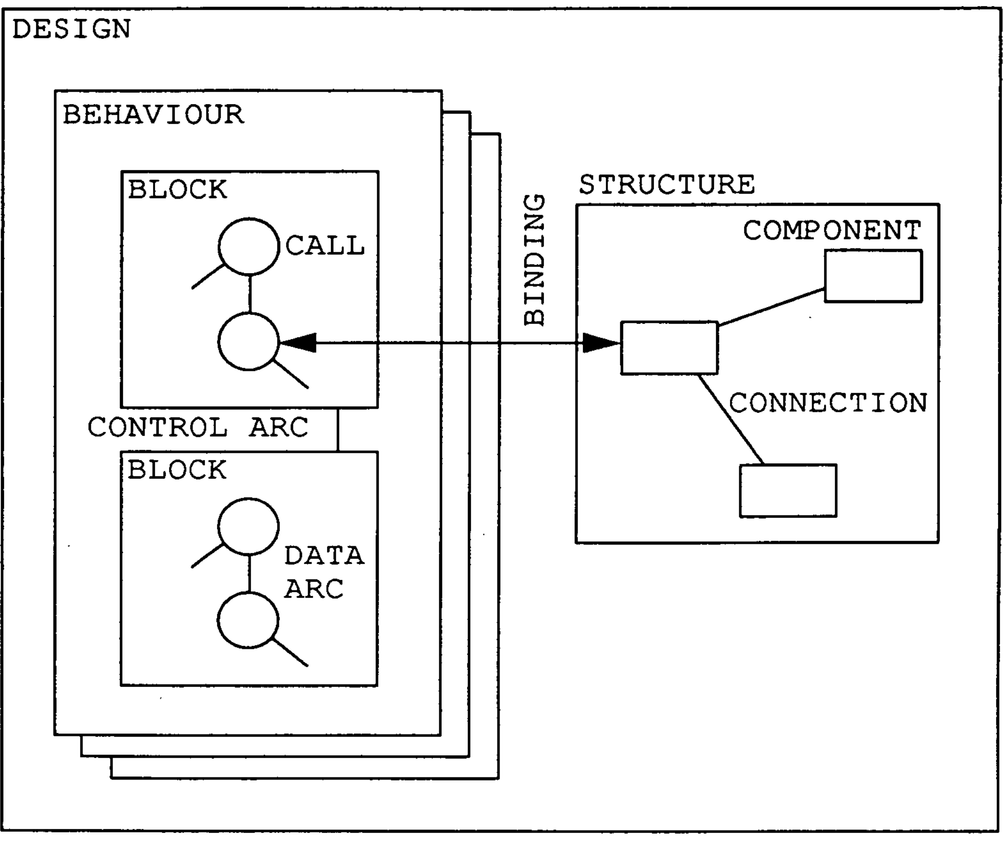


Figure 4-3. Basic SAGE ***Data Model Objects and their Interrelationships***

The time methodology that is used in SAGE is one of being single-clock synchronous, but also supporting nano-second time. Thus events can be expressed as a clock period and the delta of nanoseconds into that period. Pairs of these values can define the production and consumption regions associated with parameters of a behaviour. Thus, with negative delta's, it is possible to model inputs with a setup window requirement.

This section has provided only an overview of the basic data model objects and the time model. In fact, there are several additional concepts that are basically embellishments on what has already been outlined. In particular the concepts of zones, parameters, ports and mapping need further explanation. Zones are a subdivision of blocks, identified by calls of indeterminate length. Parameters are the value carriers that provide input and output of signals to calls and behaviours.

Mapping is a technique of sharing expensive structures across behaviours. Mapping is a complicated concept, and is addressed in further detail in section 4.8.2.3 on page 112.

4.4 Visuals

Of the objects in the SAGE database, nearly all can be identified as mathematical graph type objects, and as such it is this commonality that drives a more general approach to visual construction than would otherwise be possible. The primary exception to this, is the visual for source code, where the concepts are so different that a specific approach is required. Again, even for source code presentation many general observations can be made that leads to more flexible applicability.

4.4.1 Graphs

The main elements of SAGE graphs, are nodes and arcs. Mathematically, this would be represented as `G = (V, E)`, where `V` is for vertices or arcs, while `E` is for edges or nodes. Many textbooks and papers on the properties of these form of graphs have been written, but very little about how they can be formally mapped and manipulated for display purposes. As a simple example, consider the way arcs arrive and leave a node. Mathematically, no notion of the order that the arcs arrive is included within this graph definition. In short, many constraints and useful manipulations can be systematically defined that directly address the problem of providing a graphic or drawable form of the graph for display purposes.

Although intellectually interesting, the mathematical manipulations play only a part in graph visualisation, mainly because the manipulations are focused on the interrelations without reference to the spatial arrangement of the graph objects. The main problem with the broad thrust of mathematical treatment of graphs, is that the basic objects, namely arcs and nodes are never typed, and there is no notion of the construction of graphs. On this latter point, graph theory has not been designed to cope with the transitory state of a graph since it might be mathematically inconsistent. From the point of view of interactive synthesis there are many occasions where an intermediate stage leads to effectively an illegal graph. Such an example is having an

arc created, but no corresponding start and finish nodes that the arc will eventually start and finish from.

With the chosen workstation technology, there are several key factors that control graph visualisation. Firstly, one and two dimensional graphs are the most natural form of representation. A third dimension is not generally practical because of a lack of computing performance. The main reason for this is reflected in the facilities provided by the X Window System, which only has two dimensional drawing facilities (though recently, through PHIGS extensions, higher performance workstations recently available now support full three dimensional drawing capabilities). Nevertheless, by defining a visualisation model based on layers of drawable objects, it is possible to add an additional half a dimension. Note, this is not as straightforward as simply overlaying graphics or using graphic bit-planes, which is the simplistic approach. In many cases the two will be indistinguishable, but by having this concept, a richer set of manipulations can be supported on an unlimited number of layers. A simple example is that of removing a layer from the display, which would not be easily possible with the simple approach of a display list. This same layers concept can be augmented on the fly to represent state changes such as highlights. Using simple isometric projection, at the loss of information due to overlap, additional information by projection of values is possible. The second key factor concerning workstations is that of colour. With workstations supporting a wide range of colour options, from black and white to true colour imaging models, there is need for more complex colouring techniques than simply adopting a simple single colour. This leads to consideration about shape, defining an inside and border to a drawable object, such that the abstraction is rich enough to cope with a wide variety of graphs in a self-consistent fashion.

4.4.1.1 Graph Categories

Within this framework, combined with what is represented in the SAGE data model, general properties of arcs, nodes, and the constraints and manipulations that can be applied can be made. At the most general level, by looking at the first level decomposition of graphs, four classes of graphs can be used to categorise SAGE data

model objects. These arise from the simple observation that graphs can be directed or undirected, and in both cases, they could have the property of being acyclic or cyclic. In SAGE, blocks and their zones are basically acyclic directed graphs since they represent data-flow. Behaviour objects are again directed, but can also be cyclic. Structure objects are unusual in the sense that three options of categorisation are available, each with flaws.

The problem arises because the SAGE model recognises tristateable nodes and therefore the obvious possibility of bidirectional interconnections (as seen illustrated earlier in figure 3-20 on page 59, with an ELLA representation). Thus a structure has arcs, which for bidirectional connections are undirected, while for driver and driven port connections, the arc is clearly directed. By forcing the structure object into either the directed or undirected cyclic class, means that arcs have to have additional information associated with themselves at a lower level of abstraction. The third option is to create a graph class that understands about both directed and undirected arcs, which complicates the four graph abstraction. The engineering compromise is to select the second choice, by marking bidirectional arcs as special and consequently representing structure objects as undirected cyclic graphs.

With these four basic graph classes, the manipulations and constraints can be defined and applied in a general manner such that a rich set of operations can be developed and that not only perceived requirements are met, but, with such a general framework, new benefits can be gained. The argument is one that aims to leave as many choices to a designers creativity rather than simply limit the choices by focusing directly on the requirements as defined implicitly in the SAGE data model objects. A good example of the results of this approach is with resource-time graphs. Here there is a simple row of components along the top row, which have corresponding calls which operate on them. By recognising that the calls form one graph, and the components form another, then a more general approach to building a resource-time graph is to define two visuals rather than just one. One visual represents just the components, while the other a horizontal axis that is cross-correlated with the components visual. Thus, rather than being an end in itself, this approach immediately leads to new richer manipulations

such as for example, modifying the visual representing the components to reflect some cost attribute such as area.

Figure 4-4. Graph Categories

As well as the obvious three graphs within SAGE of blocks, behaviours and structures, a fourth general class representing only relationships in the form of mapping functions can also be defined. Maps are effectively a special case of directed acyclic graphs, where there are no reconvergent paths. In such graphs, 'many to one' and 'one to many' representations can be made. A good example is being able to map the instance names of components in a resource-time graph, to that of its type name.

4.4.1.2 Invisibility

As the stages of synthesis progress, more and more components and calls are created to reflect the results of synthesis. In the case of resource-time graphs, memory objects

appear as lifetimes. From the original resource-time graph that a user was presented with, the resource-time graph with memory components will look very unfamiliar. By making the lifetimes invisible, it is possible for a user to see the underlying function that is required to be implemented, but with the added benefit of also viewing the detailed timing. If structure graphs are also examined, the addition of memory and communication objects can easily render a schematic dominated by such elements unusable. The observation of these and other similar examples leads to the idea of invisibility.

The main benefit is that of controlling complexity, by allowing a user to selectively make nodes and arcs invisible, that are not of interest to a user during a particular stage in the synthesis. But, instead of simply removing objects, a richer set of operations that tries to preserve existing information can be made. These ideas are shown illustrated in figure 4-5.

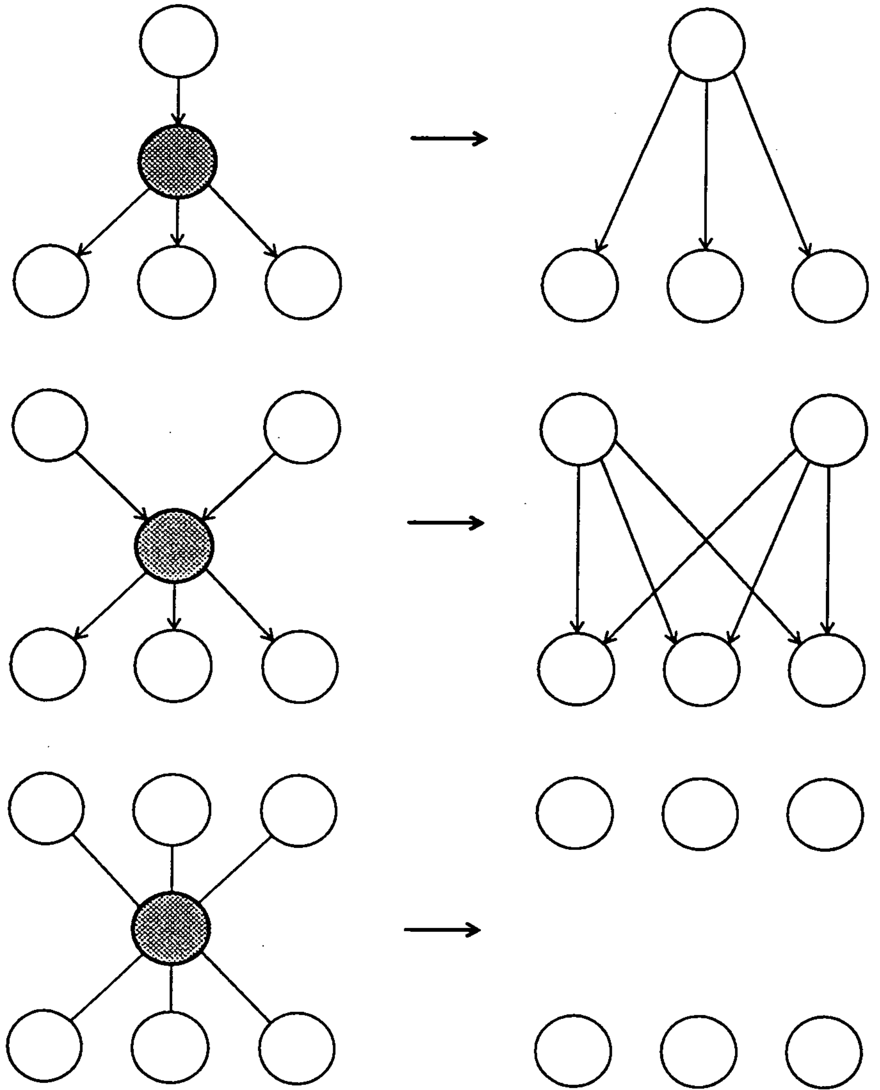


Figure 4-5. Examples Demonstrating Invisibility

The first two represent the result for directed acyclic and cyclic graphs, while the third is for undirected cyclic graphs. For directed graphs, the decision to permute the data-flows through the removed nodes is a reflection of preserving as much information about the implied information flow as possible, and more importantly the actual dependencies. In the case of the undirected graph example, at a superficial level there is insufficient information as to the exact flow of data, and therefore the normal behaviour would be simply not to bother with cross-correlating the source and destinations of all the arcs. In general, the process of making an object invisible can also determine the invisibility of attached objects. In actual operation, objects have an

action procedure associated with themselves, that determines one of two actions to take, namely cross-correlate or become invisible. This simple approach, can make control lines invisible, but leave major data flows visible in schematics.

4.4.1.3 Nodes

With a two-dimensional point viewing plane, nodes can be treated not only as simple shapes that can resemble common objects encountered in the SAGE data model, such as multiplexers and memory elements, but also in a more general fashion that means each ordinate can be used to represent different information. In the case of resource-time graphs, it is clear that the vertical ordinate can be used to represent the duration of calls, which means that the x-coordinate can be used to represent another aspect, such as cost about the call, or a certain width to represent the type of call it is. This same general approach means these ideas can also be applied to other graphs, rather than having to generate special cases. For example, in structure graphs, the x and y ordinates can represent area costs, such that it becomes very obvious at a glance which component in a schematic is expensive in terms of area. In this particular example, it is clear that the ordinates must represent the square root of the area cost in order to be representative.

Objects, whether static or dynamic need a consistent approach to defining their origin, in not only a single dimension, but both. In addition, it is not sufficient to treat objects as having only one origin, since objects like function calls need to be placed at, say, absolute time locations, whereas components can be placed with respect to an origin that is simply at the centre of the object. Figure 4-6 identifies an approach based on a three by three grid, which provides a rich set of objects, but in a general fashion. All these shapes have a defined bounding box represented by the three by three grid. The emboldened points on the grid represent the anchor points. Rather than having nine defined origins, all that is needed is a tuple, with each element consisting of one of three values to specify the origin. This is very similar to the ideas used for text representation (as shown in figure 5-14 on page 143), but in that case, the issue of rotation is also important, and has therefore been considered.

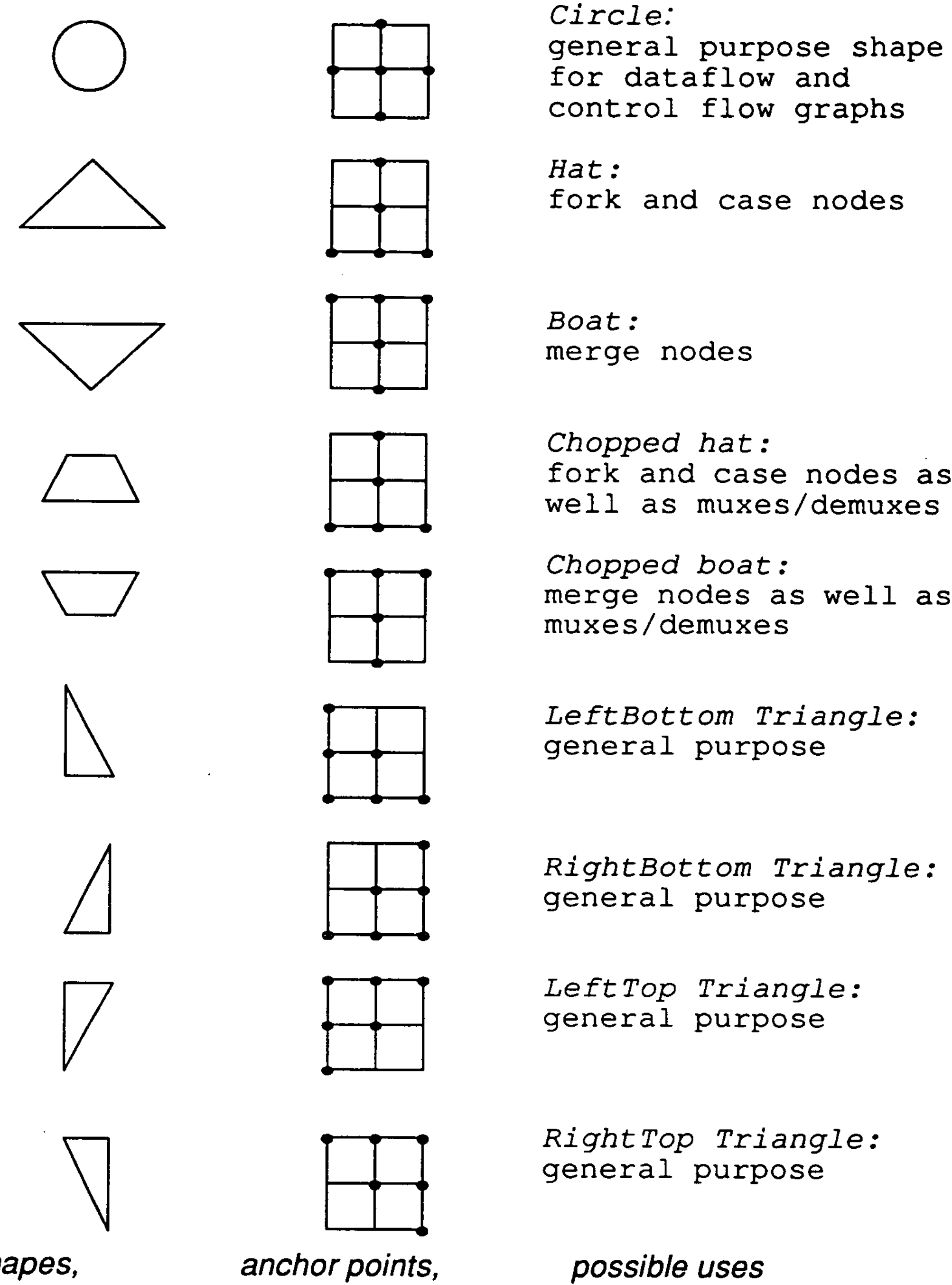


Figure 4-6. Shapes and their Anchor Points

With static objects, the specified anchor point defines where any connected arcs will generally arrive. From a visual point of view, an arc visibly terminates at the boundary of a shape, but the natural extrapolation of the line is towards the anchor point. Where this treatment changes, is when the arcs are routed using Manhattan lines.

Dynamic objects, which have some ordinate specified as some operational characteristic of the design, need special treatment only in the case of objects used in resource-time graphs. Nevertheless, even for resource-time graphs, adopting the general approach can lead to fairly faithful reproductions of resource-time graphs. There are several reasons why it is still inadequate to support properly resource-time objects. The two most dominant reasons are the presence of indeterminate length calls, and the fact that calls can be pipelined and therefore a simple rectangle representation is no longer possible.

The general problem of representing resource-time graph call objects is solved by using a parallelogram. This can be compared with earlier solutions such as used in SAGE 2, where pipelined objects were represented as upside-down flower pots. The main problem with this approach was that overlapping of objects would soon appear confusing, especially if multiple pipeline executions were in progress. Figure 4-7 highlights the difference. As well as being clearer to see pipelined objects, such operations now naturally degenerated to rectangular boxes when the call is not pipelined. Additionally, since both vertical sides are now genuinely vertical, they can be marked along their length to represent indeterminate length operations as well a provide consistent locations for arcs to arrive and leave. The important point is that this happens without having to make a special case of pipelined calls. The detailed issues of representation of resource-time graphs are explored late in section 4.8.2.

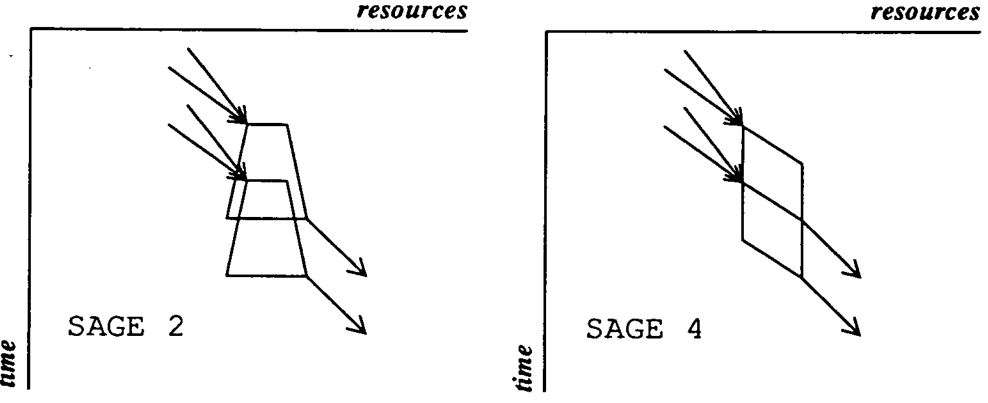


Figure 4-7. Pipelining Representation in SAGE *4, compared to in* SAGE *2*

4.4.1.4 Arcs

Three issues relate to the general representation of arcs. Namely, how they actually enter and leave a visual, how they enter and leave a node and thirdly, how they are visually represented.

For a directed graph, normal signal flow conventions are to present a user with up/down, left/right signal flow. In the case of the graphs used within `SAGE`, all arcs start and finish on what can be termed terminator nodes. As a result, not only is the terminator node usually represented differently, but in the general case, arcs that have no terminator can simply be sorted to start at the top/left of a graph if it is an input, and bottom/right if it is an output. For undirected graphs, the decision is based on what the type of port is being driven. For example, in the case of schematics, it is natural to direct bidirectionals to the bottom edge, such that they are filtered out naturally from the other explicit inputs and outputs. The issues concerning entering and leaving a node have been treated earlier in the section on nodes.

As concerns the actual method to display arcs, of three options, namely, straight lines, manhattan lines and splines, only the former two are reasonable, since they are fully consistent with the graphics capability of the `X Window System`. (Although bezier routines are available, the extra computational expense compared with the neater graphs that would result meant it was deemed a non-essential feature.)

4.4.1.5 Constraints

Where as the positioning of arcs is predominately by where the nodes are placed, this is obviously not the case with nodes themselves. Again, rather than taking a specific approach to just saying objects can be ordered as they are scanned, a more general look at the problem leads to a richer set of options in representation. Clearly, each axis represents some feature of the design. It could be simply, time, it could be alphabetical sorted objects, it could be all objects of a certain type or some other function related to the graph being displayed. These observations lead to the idea of an ordered axis with its opposite of an unordered axis.

Of the latter, which is the less complex of the two, if a graph axis is unordered, then, the elements can be arranged in manner of most convenience to the graph generation algorithms. One special concept that applies to an unordered axis is that of being able to collapse all the nodes, such that they coexist in the same column (or row). As will be seen, this is effectively the same as an ordered axis with the directive of placing all nodes in the same column (or row). Clearly, such ideas can be abused, as in having both axis unordered and directed as collapsable. But, from this general approach, the problem of displaying critical paths in a succinct form becomes much simpler, since single thread critical paths can simply be highlighted and the highlight collapsed since there will be no significant overlap. For simple graphs, this is not so important, but when chasing a critical path up and down hierarchies, it becomes more important to be able to extract that information and present it in this fashion.

Constraints involving time are more complex. Along an ordered axis (say in the x-axis), there are implied columns of information. How the columns are spaced is one issue, and how information is assigned to the column in relation to time, is the second issue. This approach handles coarse clock tick level time automatically, but also means more general protocol defined transactions can also be handled automatically. The classic example of this is viewing a resource-time graph in timesteps rather than in actual clock periods. In this case, the abstraction means that complex problems raised by the introduction of indeterminate calls can be easily bypassed, since all calls, whether indeterminate or not, are assigned a unit height, rather than the normal behaviour of a resource-time graph of assigning a height in relation to the actual duration of a call.

4.4.2 Text

Text graphs provide valuable cross-correlation facilities with design objects represented in the SAGE data model. Since a designer's main point of input is in text, whether VHDL [31] or ELLA [34], being able to relate objects back to the source code helps a designer navigate and therefore direct the synthesis process in a way that should achieve design objectives much easier. But since input source text is not the only text that a designer encounters, a more general approach leads to a framework

with wider applicability. In particular, the three other main usages of text are transcript output, generated netlists and the help facilities. While they appear to be just simple interfaces, and many tools provide only the simplest of text interfaces comparable with editors like 'vi', a closer look reveals much more opportunity to define concepts that empowers a user to a much greater degree. A fifth text area could also have been considered, namely that of command line input, and this would lead to the natural, but very powerful concept of a universal editor, of which the most significant example is that of the APOLLO Display Manager editor.

All the text regions have two significant facets. Firstly, they have, as with many other languages, well defined grammar and secondly well defined lexical semantics. In the case of the transcript output and help facilities, the grammar can be considered to be virtually non-existent, while for languages like ELLA and EDIF [33], complex (in compiler language talk) non-terminal objects can be identified.

Decomposing text based on grammar, results in page/row/column tuples to identify regions of text. Note, how these figures are produced is not important, but the fact that no 'on-the-fly' parsing is needed to identify these elements helps efficiency. Clearly, this is a sensible division since the complexity of an on-the-fly compiler would outweigh any benefit that it would provide in identifying text regions. In the case of the text input into the SAGE system, this process of identification is done by VTIP [45]. (The only flaw with VTIP is that the tuples have no notion of column number, i.e. only a line by line basis is supported). In a similar fashion, the netlist generation routines can produce additional information that correlates non-terminal objects in the output with SAGE objects. The point to note about these examples is that the stage where the tuples are generated are not related to the parsing process of the language, but more to when data-structures representing the language are being traversed in order to produce output.

With 'help' text areas, tuple pairs will tend to point to identifiers. Rather than the tuple pair referencing a SAGE data model object, it would be normal to expect it to reference further help. This facility of cross-correlating information in this manner is usually termed hypertext - a sort of read-point-click cycle.

Examining the lexical tokens that can be encountered, five useful categories can be identified that can help a designer analyse the structure of any text viewed with greater ease. These are keywords, quoted strings, comments, numbers and the remainder not including whitespace. Unlike the tuple pairs for text regions, these five represent a disjoint set. An examination of languages like VHDL, ELLA, EDIF and ADA, shows how fairly straightforward it is to identify these items. A more complete breakdown is possible, but this set provides a useful set. In fact, considering some of the commonest problems of losing valid code as a result of not terminating strings or comments in languages like C, it is surprising that no editors support simple highlighting like this as a natural part of the text editor interface. Figure 4-8 gives an outline specification of a lexical analyser that would recognise and highlight the elements in the ADA language The format is in lex format [6]. Note, it has been written in a form so it should be clear how it works and therefore is not a complete or efficient representation of what is required.

```
...
%%
... /* all the ADA language keywords */
{begin}  |
{end}    |
{return}        start_highlight_keyword();    ECHO;  end_highlight_keyword();
"[^"]*"         start_highlight_string();     ECHO;  end_highlight_string();
--.*$           start_highlight_comment();    ECHO;  end_highlight_comment();
[0-9]+          start_highlight_number();     ECHO;  end_highlight_number();
(" \t\f\n")*                                  ECHO;
.               start_highlight_remainder();  ECHO;  end_highlight_remainder();
```

Figure 4-8. Highlighting Text Items

4.5 Drawing Directed and Undirected Graphs

Approaching the problem of laying out a general graph, various observations about the difficulties involved can be made. The most important aspect to this, is that of aesthetic appeal, which is, by its very nature, subjective. Graph theory has provided many rules for planarity checks and as mentioned earlier, many mathematical properties, but very little about how graphs actually should look. The most significant contribution to this area can be found in [59], where many attributes of graphs are defined, as the goal of layout becomes the task of optimising each value depending on a formula. Clearly this results in various heuristic algorithms that are generally iteratively applied after evaluation of the cost attributes. In many ways it is a

microcosm of what the general approach about interactive synthesis achieves, but in this case, all the steps are automatic.

What is presented next are the heuristic algorithms developed that focus on the primary problem of taking a general graph, G= (V, E), and simply aiming to provide top down signal flow (which could just as well be left/right signal flow), and following the cost measurement of arc cross-overs as being the primary measure to minimize. The most complex aspects are related to identifying the loops in the graph, and handling what appear as simple details like drawing the arrow heads on the ends of the arcs when they meet the nodes.

4.5.1 String Method for Breaking Loops

In order to obtain top-down signal flow, the nodes need to be placed one after another, following the signal flow as closely as possible. If there are no backward flowing arcs, then there is no problem. In fact, if all nodes are assigned a unit height, then the process with acyclic directed graph is very similar to one of as soon as possible (or ASAP) scheduling. The difficulty is with cyclic directed graphs, which interfere with the general top/down signal flow.

The methodology adopted to solve this problem is comparable to walking around a graph with a piece of string, tying it to each node that is visited, and as soon as a node is visited that has already been visited previously, then one of the arcs that flows between where the string leaves the re-visited node, to where it re-enters needs to be marked as having been effectively reversed. The tricky bit is to recognise how far to backtrack to ensure that the algorithm remains efficient. In brief, if the last arc is effectively marked as being reversed, then the algorithm can simply backtrack one node and continue searching the graph, using the same heuristic. The neatness of this algorithm ensures that even loops within loops can be simply handled. The second necessity for ensuring maximum efficiency, is marking the nodes that have been visited only when the string is finally removed from it. This means that nodes that have been backtracked from because they form a loop, must not be marked, since the string still passes through them. Because of the way the algorithm works, if these

nodes are not marked during the backtrack stage, the algorithm will still work, since the decision to backtrack is based on encountering effective dead ends.

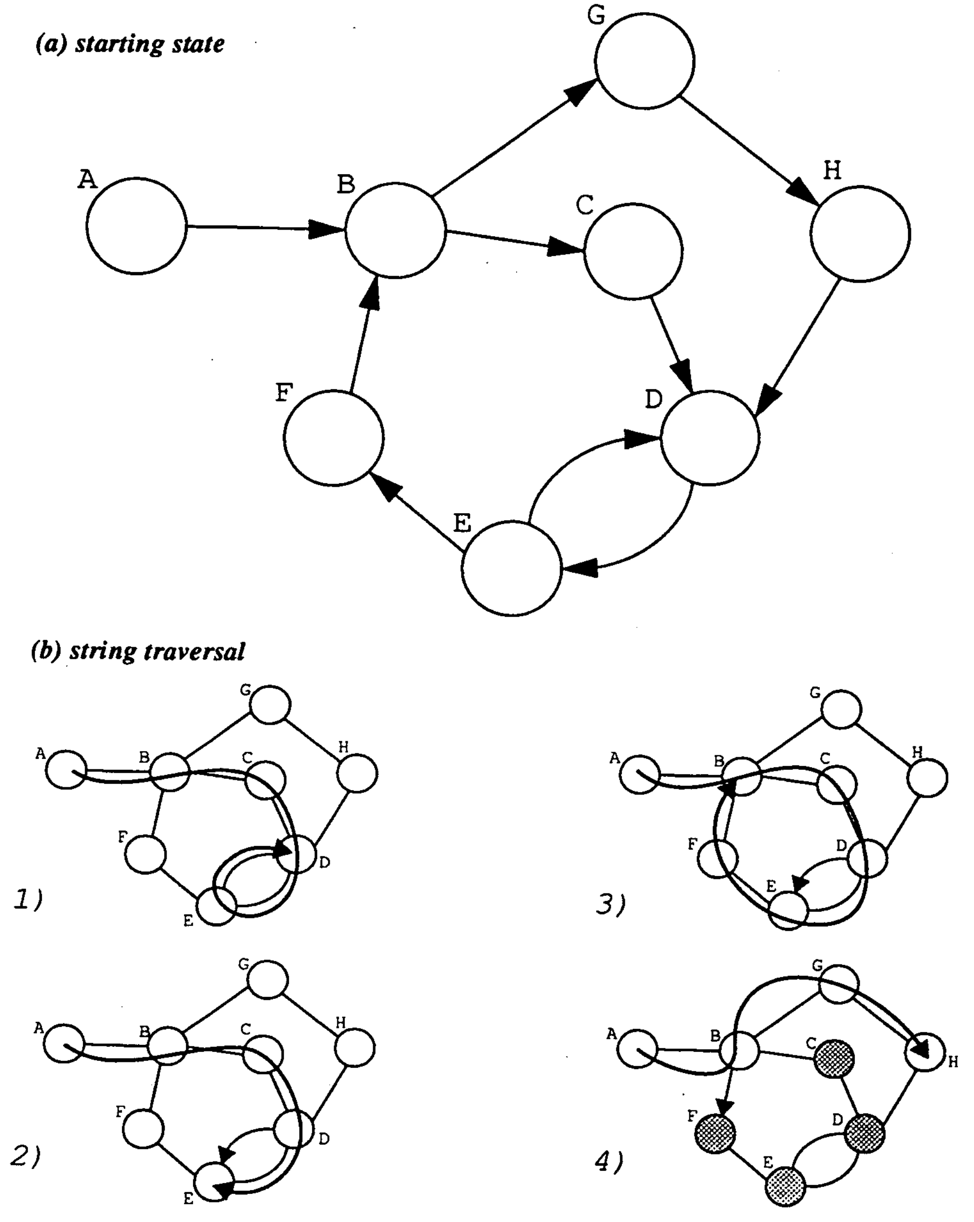


Figure 4-9. Example to Illustrate the String Method

In figure 4-9, there is an illustrative example of this algorithm in operation. Part one in the figure shows how the string has followed the forward arcs to node E, where it had a choice of two outward arcs to follow, and for this example, the arc referring to D is followed. This means the string has come back on itself, and therefore the last arc encountered is effectively flipped around. The string then backtracks to node E, as shown in part two, and makes its way forward to node B, where again the same action

is taken, as shown in part three. When the string is at F, the process of marking nodes that have been fully visited starts, in that, F, E, D, and C get marked as being fully visited, as shown by the shading in part four. With the string back at B, the second remaining outward is followed, and it will soon encounter the marked node D, and therefore begin backtracking all the way out of this graph.

Because of the way it works, the algorithm also handles graphs that are already full loops - in the sense there is no clear starting node to where the algorithm should be applied. As a result of this property, the algorithm can be applied to all nodes in a given graph, provided the node has not already been marked. An additional benefit of this is that graphs which consist of disconnected graphs are handled automatically without having to select them as special cases. Another case of not needing a special treatment, is that of self-referencing nodes, which automatically have such arcs flipped around. For cyclic undirected graphs, this same algorithm can be used to remove loops, by assuming that all arcs that have been visited for the first time are outbound arcs. In the case of structure diagram, this property can be improved by applying this behaviour only to bidirectional lines.

The final piece of information that needs to be attributed to nodes, is their position. At a simple level, it would seem sensible to make it simply the length of the string up to that node position, but as with ASAP scheduling, the node must move to a position that ensures all input arcs into that node flow top to bottom. This requires that on the forward pass, the current string position is placed in any unmarked node. But, for marked nodes, the maximum of the last string length and current string length is placed in the node. This means that as a marked node is successively visited it, it will either stay where it was placed, or be gradually floated downwards.

The neatness of the algorithm is highlighted by how no matter what the nature of the graph, that is if it consists of disconnected subgraphs and/or complex loops the locations are determined in just one pass with no special cases. The only simplification has been to have no heuristic to decide on which arc to actually flip. This weighed against the objects in SAGE that do generally exhibit top down information flow and the fact that they do have well defined entry nodes to the graph

to which the algorithms can be initially applied, means that in the most part, this choice of arc to flip will produce sensible graphs.

4.5.2 Barycenter Application

Having identified where the nodes have been placed vertically, it is next necessary to place the arcs. Simple point to point drawing is highly undesirable, because of the overlap that can easily result. As a result, for an arc that starts from row n, and finishes at row n+m, then space needs to be allocated for the passage of the arc at rows n+1 to n+m-1, provided this represents an ascending sequence. The only exception is when m = 0, in which case space needs to be allocated for what is effectively a local loop. This space allocation is illustrated in figure 4-10. In principle, the width allocated to a passing arc can be anything, but a width equivalent to the effective widths allocated to nodes has been used. When comparing this approach with other routing algorithms, its relative simplicity can be seen as a consequence of no limit on space consumption in the x axis.

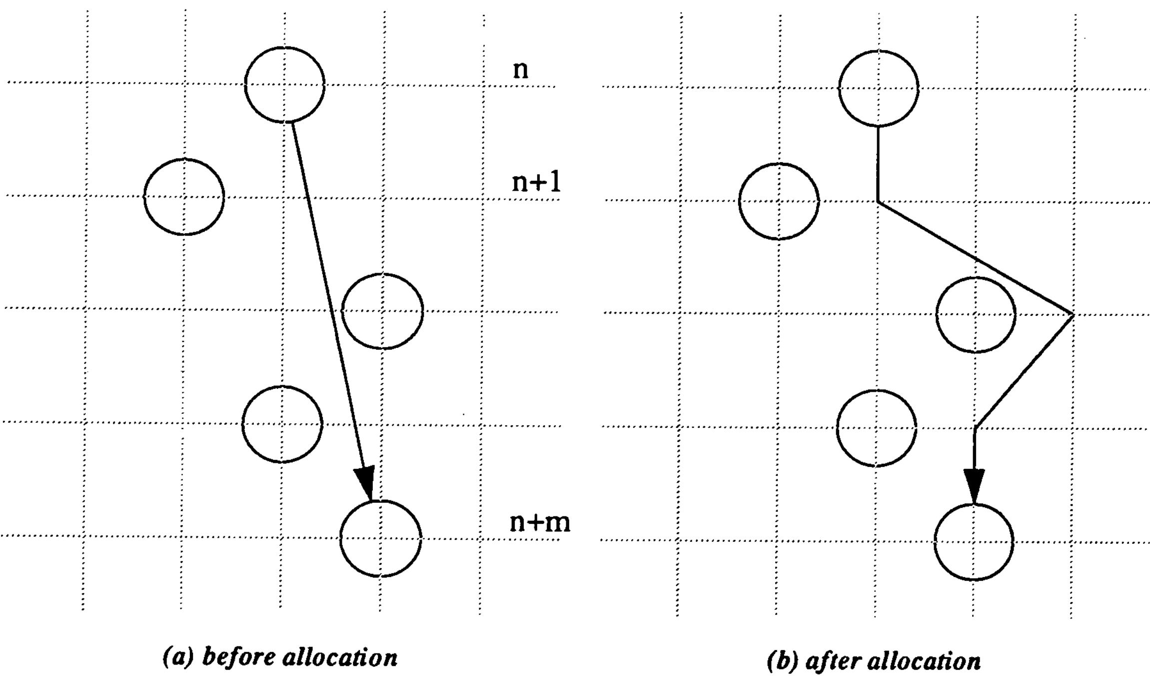


Figure 4-10. Space Allocation for Arcs

The routing of these arcs does not need to be clever, since the next stage is iteratively to move these arcs and nodes in each row, such that a more aesthetically pleasing

graph is produced. The approach taken, is that termed the barycenter approach, as identified in [59].

In that paper, it is described how Carpano proposed an iterative method for the reduction of crossings in a two-layer graph, called the 'relative degree algorithm'. This meant that given a placement of the vertices on level 1, the abscissa of each vertex on level 2 is computed as the average of abscissas of its neighbours on level n called the up-barycenter. Then, a symmetric step is performed by computing the down-barycenter for the vertices on level 2. The extension to the n-level graph follows naturally, but with the additional requirements that edge conditions on the first and last layers are handled carefully and the possibility of computing the weights across more than just the preceding and succeeding layers arises.

The algorithm effectively splits the difference between the current node location and that suggested by the arcs above or below, that are actually pulling on the node. As a result, the algorithm moves the nodes to a location that eventually minimizes the energy represented in the graph. Because of the granularity of the placement allowed, and the fact that the algorithm only examines a layer at a time, the total energy cannot not be certain to minimal, and in fact, complex oscillations in the node positions can result when coming towards the end of a number of oscillations. As a result, although a rough number of forward and backward iterations can be estimated as having to be at least $2*\log_2(width)$ to allow for the worst-case migration of a node, because of the oscillations, the exact value used is not important.

4.5.3 Arrow Heads

One minor issue related to drawing of graphs is that of drawing arrow heads on arcs. Most drawing packages avoid this problem or use a bitmap to represent the end of an arc, and that based on a simple approximation of the final angle of the line. The two aspects discussed here are the transformations needed to draw an arc abutted neatly to

a circle, and then of finding the angle and point of intersection between a circle and an ellipse for the special case of local loops associated with nodes.

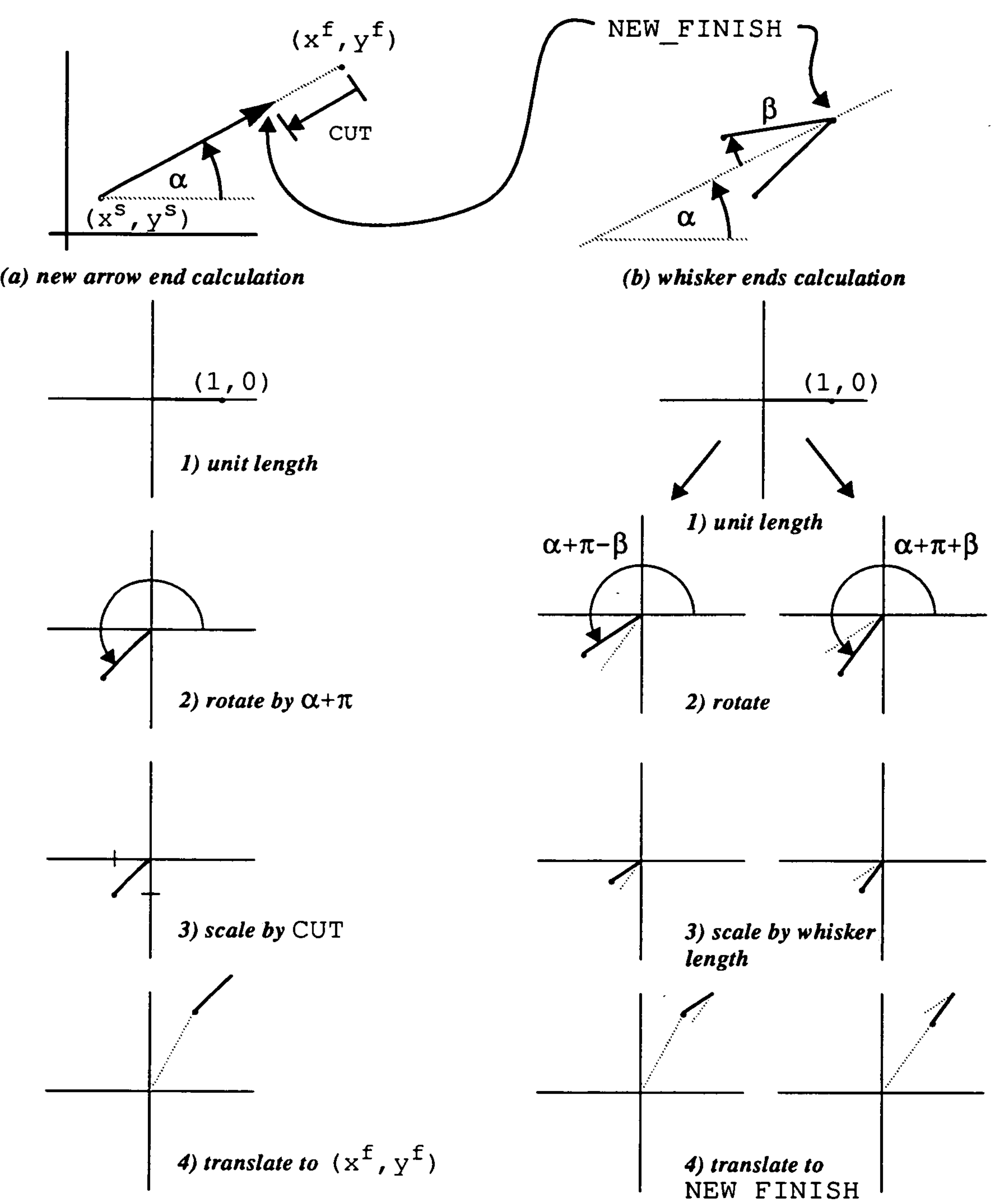


Figure 4-11. Transformations for Arrow Heads

In figure 4-11, the various stages that have to be followed are illustrated. Given a vector (x^s, y^s) to (x^f, y^f), and a radius representing the circle of intersection,

three points need to be computed, namely the starts of the two whiskers and the finish point. Given that the line makes an angle of α between the horizontal, then the four steps needed to identify the finish point are shown in part (a) of the figure. In a similar manner, the two whisker start locations are computed by a rotation, scaling and translation step as shown in part (b). The actual whiskers are drawn by simply joining the computed points.

The issue of local loops, when solved by using the path of an ellipse, slightly complicates the problem by requiring the point and angle of intersection to be computed. The angle is obtained from the derivative of the ellipse at the point of intersection, while the actual points of intersection are found by solving two simultaneous equations.

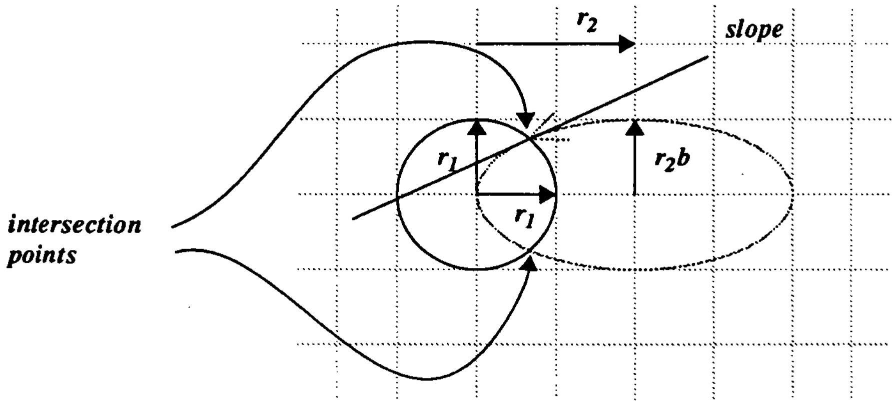


Figure 4-12. Local Loops

Figure 4-12 illustrates the problem. The equation for the first ellipse, which is always a circle of radius r_1, is simply: $x^2 + y^2 = r_1^2$. The equation for the second ellipse is: $(x - r_2)^2 + \frac{y^2}{b^2} = r_2^2$, where the vertical radius is related to the first circle, by the relation: $(r_1 = r_2 b) \Rightarrow b = \frac{r_1}{r_2}$. Differentiating the equation for the second ellipse, and rearranging produces: $\frac{dy}{dx} = \frac{b^2 (r_2 - x)}{y}$, which can be used to determine the

angle of intercept. The two ellipse equations are effectively two simultaneous equations, which can be subtracted form each other and rearranged as a quadratic in x: $(b^2 - 1)x^2 + (-2b^2 r_2)x + r_1^2 = 0$. Using the normal quadratic equation solutions: $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$, such that $a = (b^2 - 1)$, $b = -2b^2 r_2 = -2br_1$ and $c = r_1^2$, will give the possible solutions as: $x = \frac{r_1 (b \pm 1)}{(b+1)(b-1)} = \frac{r_1 r_2}{r_1 \pm r_2}$. For the example figure given, only the case for $r_2 > r_1$ applies. For this case, substituting back into the circle or ellipse equation, provides the remaining y intersection points as: $y = \pm \sqrt{\frac{r_1^3 (r_1 + 2r_2)}{(r_1 + r_2)^2}}$.

4.6 Inter-Visual Interaction

As well as information within a graph visual, and information available by cross-correlation highlighting, there is additional inter visual association that can be made. What appears a simple concept, is in fact very powerful and solves numerous practical display problems as well. The idea is that for a given axis, the selection method chosen also drives the selection of the objects on the corresponding axis of the new attached graph.

By way of example, consider the use of a resource-time graph where the x-axis represents the structural components, and the corresponding calls that are active on the structural components, if any. This defines an order and a set of objects, that can form the basis for another graph. The x-axis of this other graph is effectively tied with the way the x-axis on the resource-time graph behaves. In this second graph, the flexibility arises in the second axis (y-axis), where there is now a choice to be able to represent a quantity other than time.

If the two visuals are physically placed next to each other, then the two axes can be considered to be hard-tied, and there is an implication that any panning operation in

one of the graphs must be reflected in the other graph to ensure that the alignment is not lost. The other form of tying is that of being soft, where not only is panning not preserved across the graphs, but neither is there need to keep the new axis in the same vertical or horizontal orientation as the one it is tied with. For example, in figure 4-13, the x-axis is soft tied to a y-axis, which is used to represent a set of three visuals that are hard tied in the y-axis.

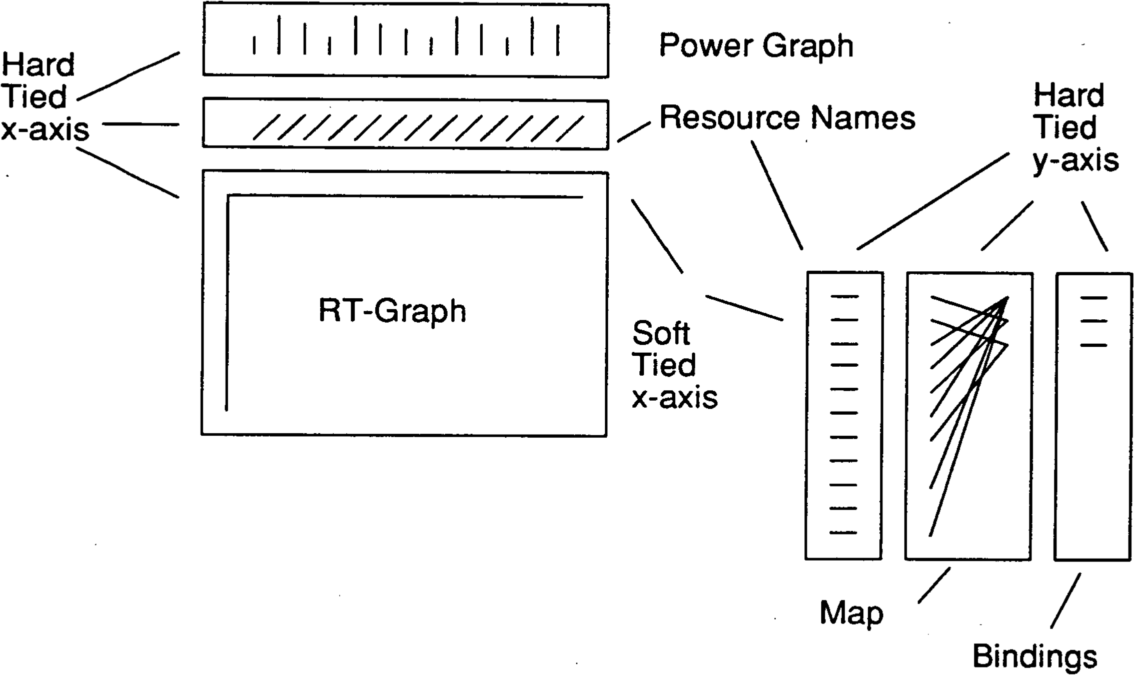


Figure 4-13. Example Illustrating Hard and Soft Combined Axis

In this figure the x-axis of the resource-time graph is used to drive three other graphs. In the first case, we have identified the resource names and a bar chart representing their current area cost. The mapping shows how the resource-names map onto the actual type names of the resources.

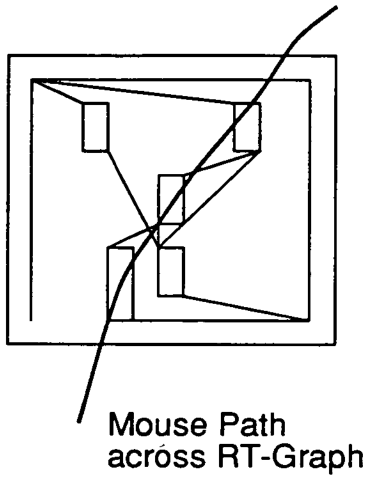
As well as empowering a user to associate different facets of information represented in the data model, the user also gains in ease of use by having distinct pannable screen regions. A good example is with the resource name to type name mapping, where a user could find a string clipped by the window, and is happy to pan, provided the panning does not lose important information. If all the three graphs were present in the same visual, then a pan operation for a string in the type column, could lose the

resource names from the users field of view. Especially with the problem of strings of indeterminate length, this approach also means it is no longer necessary to truncate arbitrarily the strings to prevent obliteration of possibly important design information, since they can simply be panned into view as necessary. This is only a partial solution, since text names associated with objects within a resource-time graph, need to be still treated in a manner that ensures the graph does not become overly drowned by a sea of text. The simple solution of just controlling the text layer visibility and other related issues are examined later.

4.7 Dynamic Visualisation

A user examining a graph is presented with what is essentially static information. In a complex graph, there are many interrelations, of which a designer will be focusing on only a few. A good analogy is with chess, where a chess player focuses on a small region of the board, and sees how the pieces directly in view, can now and then relate to other pieces at the far corners of the board, by animating in the players mind various moves. In this same localised way, a designer would analyse graphs to determine the next synthesis action.

In these cases, it is simple to recognise simple rules that can visually distinguish key aspects of graphs that are of interest to a designer. The important point is that to be in keeping with the spirit of a designers thought processes, the results of this highlight have to be fast enough to allow a designer to move rapidly to another region of a graph to follow a hunch about what actions the designer might take next. This process is called dynamic visualisation, and is defined as some graph highlight action that is a function of where the pointer of a mouse currently resides. In the example shown in figure 4-14, the action is defined as one of highlighting a node and any of its related arcs, when a particular node is pointed at. The complexity of the action then becomes a function of the performance of the hardware and the average delay between successive highlights that a designer will be happy to tolerate. Issues like this are also affected considerably by the algorithms used, and therefore care has to be taken in the development of these algorithms. This algorithm performance issue, and many others are explored in detail in chapters 5 and 6.



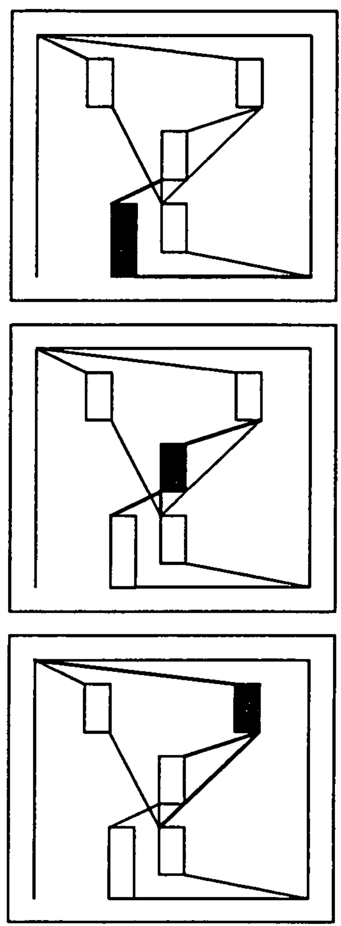


Figure 4-14. Dynamic Visualisation in Action

4.8 Visuals in Action

The following sections enumerate in detail, all the visuals that have loosely followed the concepts that underlay graph presentation. The four major interfaces are the display facilities for source code, resource-time graphs, control-flow graphs and structure. Since each graph also has many objects that relate to other graphs, their are significant issues relating to cross-visual highlighting that are addressed in the section on interaction. Also in the section on interaction is the issue of interactive synthesis actions that can apply to certain data model objects.

This sections focuses on the key display characteristics, and SAGE data model concepts that are captured rather than every detail that is associated with the visuals. A large number of the general display issues have already been covered in section 4.4.

4.8.1 Source Code

The main feature of the source code visual, is the way it is limited by restrictions imposed by the VTIP model. This means that objects of interest are only specified by line numbers, and therefore references to multiple objects on the same line can cause confusion to a user, since there is no way to distinguish them. The approach taken is to provide three vertical regions that can represent the three types of object that can be identified in the source-code and correlated back to the relevant objects in the SAGE data model. These objects are 'calls', 'blocks' and 'behaviours'. Within each region are vertical lines that represent the tuple over which the highlight happens. Clearly, because of the granularity of the tuple, many call arcs can reference the same region, but it would be unusual for this to happen with behaviours and blocks. Another observation, is the way that text regions for blocks and behaviours might form disjoint sets, particularly since the source code compilation process might do some clever decomposition/optimization and find itself merging disparately related basic blocks.

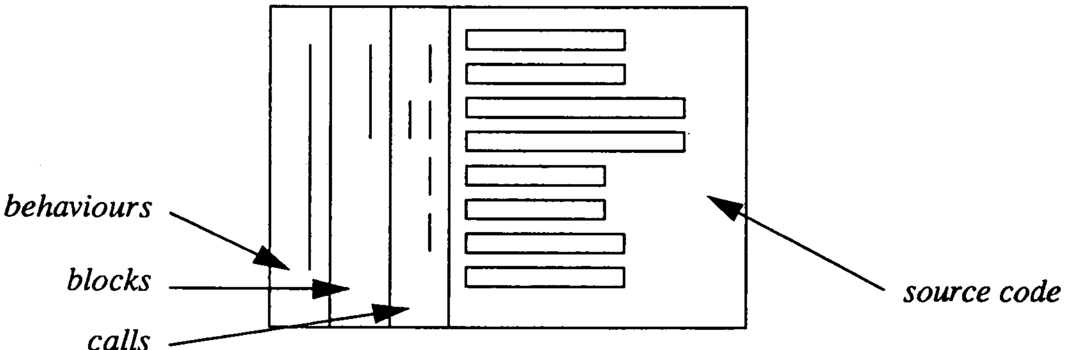


Figure 4-15. General Layout of the Source Code

The figure above is the general form of the source code visual. The first three columns represent highlight marker regions for behaviours, blocks and calls. The region on the right is the container for the actual source code.

4.8.2 Resource-Time Graphs

Of all the visuals, the resource-time graph object is the most complex. In some ways this reflects the primary position it has as an interface with which a user is able to make design modification decisions, compared with the other visuals. Note, this does not mean that the other visuals provide anything less in terms of general interaction capabilities, but more that there is less scope for interaction with these other graphs because of the nature of the information that is presented. Why the resource-time graph holds this position is because it captures, in its raw form, two of the most important attributes that a design exhibits, namely resource and time.

Resource-time graphs in conjunction with control-flow graphs capture the main operational characteristics of a design. With control-flow graphs, the designer can see the passing of the thread of control in a top-down manner, with any looping or branching illustrated by arcs that generally flow backwards. Resource-time graphs, being acyclic, only have arcs flowing forward, representing the passage of data-items from one function call to the next. As will be seen discussed in the results chapter 7, manipulations at the control-flow graph level can provide the most significant synthesis actions.

4.8.2.1 Zones

Although all the calls local to a behaviour form part of the resource-time graph, they themselves can be further categorised into zones. This concept is necessary both for scheduling purposes and display purposes, because of the difficulty of handling indeterminate length calls. By definition, a zone is defined as a group of function calls within a basic block that can include indeterminate function calls only at the output. There are three situations in which such indeterminate calls can arise. These are library components of indefinite length, loops with indeterminate iteration counts and branching statements whose branches have different (i.e. not necessarily indeterminate) length.

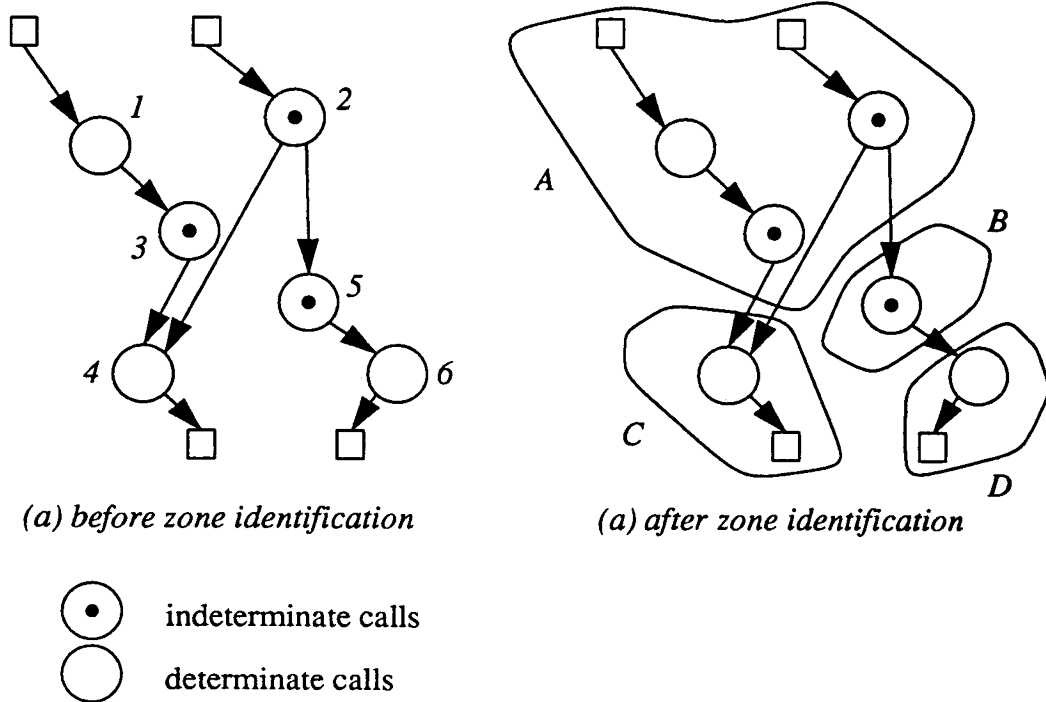


Figure 4-16. Zone Identification Example

For a given behaviour, there are generally many ways to identify the zones. Figure 4-16 illustrates how zones are identified for a given data-flow graph. With the indeterminate functions defining definite zone boundaries, the remaining zone identification process aims to place as many calls in each zone. Thus zone A could have been identified as two zones, but has instead been collapsed into a single zone.

In principle, zones could operate in parallel, but for simplifying reasons, they are forced to operate in series. One of the main benefits of this simplification is the controller, which needs only to produce control marks to initiate the next zone, when the previous zone has indicated that all its indeterminate operations have completed. Whereas it might appear that this simplification can help the visualisation software, this is not the case since the controller software should add in calls that start relative to another call, as opposed to starting relative to a zone. Early SAGE versions did not need to consider the issue of indeterminate length operations, since all operations were defined as being of a known duration. The next generation of resource-time

graphs provided only one zone in one graph. The latest generation now display all the zones, which has the added benefit of showing the inter-zone data flow, which is of the same importance to a designer as the intra-zone data flow. For the example given in figure 4-16, the following figure 4-17 shows the form of the resulting resource-time graph.

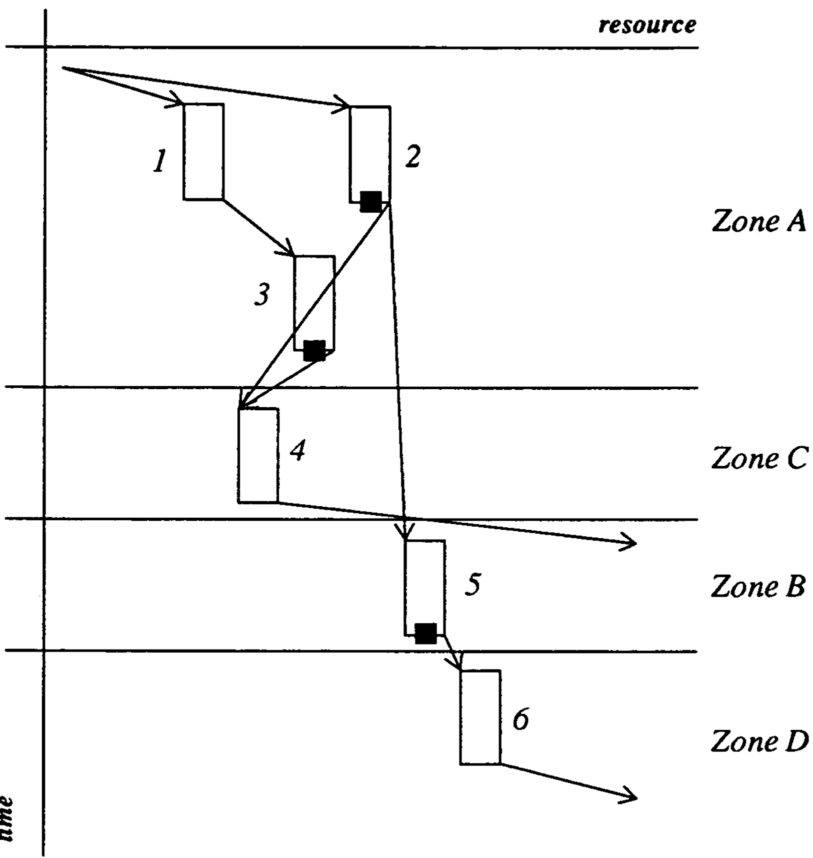


Figure 4-17. Zones in a Resource-Time Graph

4.8.2.2 Call Shape

Since calls are effectively instantiations of behaviours, their temporal complexity is directly related to the complexity of their behaviour. This shape is governed directly by the class of component, which is usually reflected in the parameter behaviour. Calls can be clocked or unclocked, and those that are clocked can be of indeterminate

length. This is illustrated in figure 4-18 with the ticks (✓) representing the supported combinations.

	determinate	indeterminate
clocked	✓	✓
combinatorial	✓	

Figure 4-18. Supported Call Classes

As illustrated in figure 4-19, if hierarchy is considered, there are implied restrictions on the calls, if any, of such a call. This information is needed when analysing the shape of a call, if it is hierarchical. For example, the calls of a clocked/determinate call can consist of only clocked/determinate or combinatorial/determinate calls.

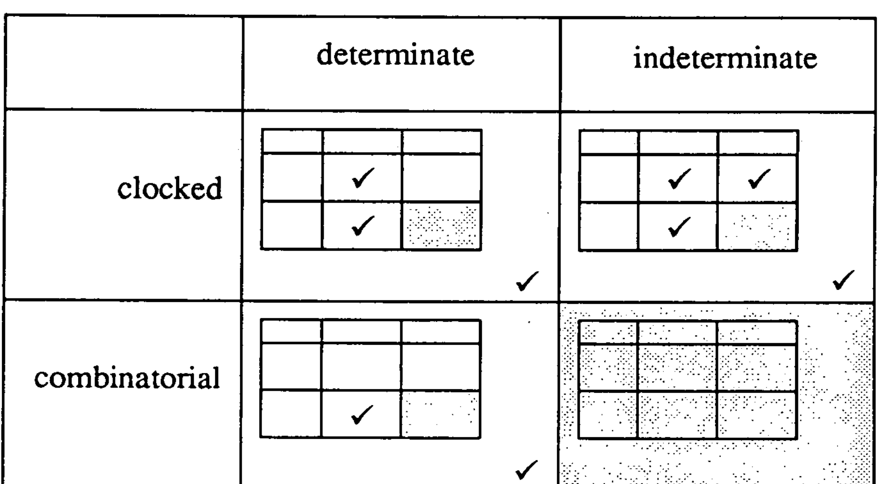


Figure 4-19. Call Classes and their Children

Both clocked/indeterminate and clocked/determinate can have two groups of parameters in common, both of which are associated with definite clock tick marks.

These two groups are simply the input and output parameters. In the case of indeterminate function calls, there are again two groups of parameters, but this time, they cannot be formally associated with any time-mark. For display purposes, they are placed at the end of a call, separated by a clock tick, with the existence of these indeterminate input and output parameters clearly marked by a small rectangle placed at the bottom of the call. The structuring of an actual call, is shown in figure 4-20. All inputs appear on the left, and outputs on the right. When there are multiple inputs and outputs happening at the same time, they can be separated visually. If the parameter spacing is set to zero, then the parameters themselves are not displayed. This means if parameter offset is also set to zero, then the calls will look very much like earlier versions of the resource-time graph. Notice how the flange concept goes one stage beyond this, in actually sorting the signals by the zones that they go to or come from.

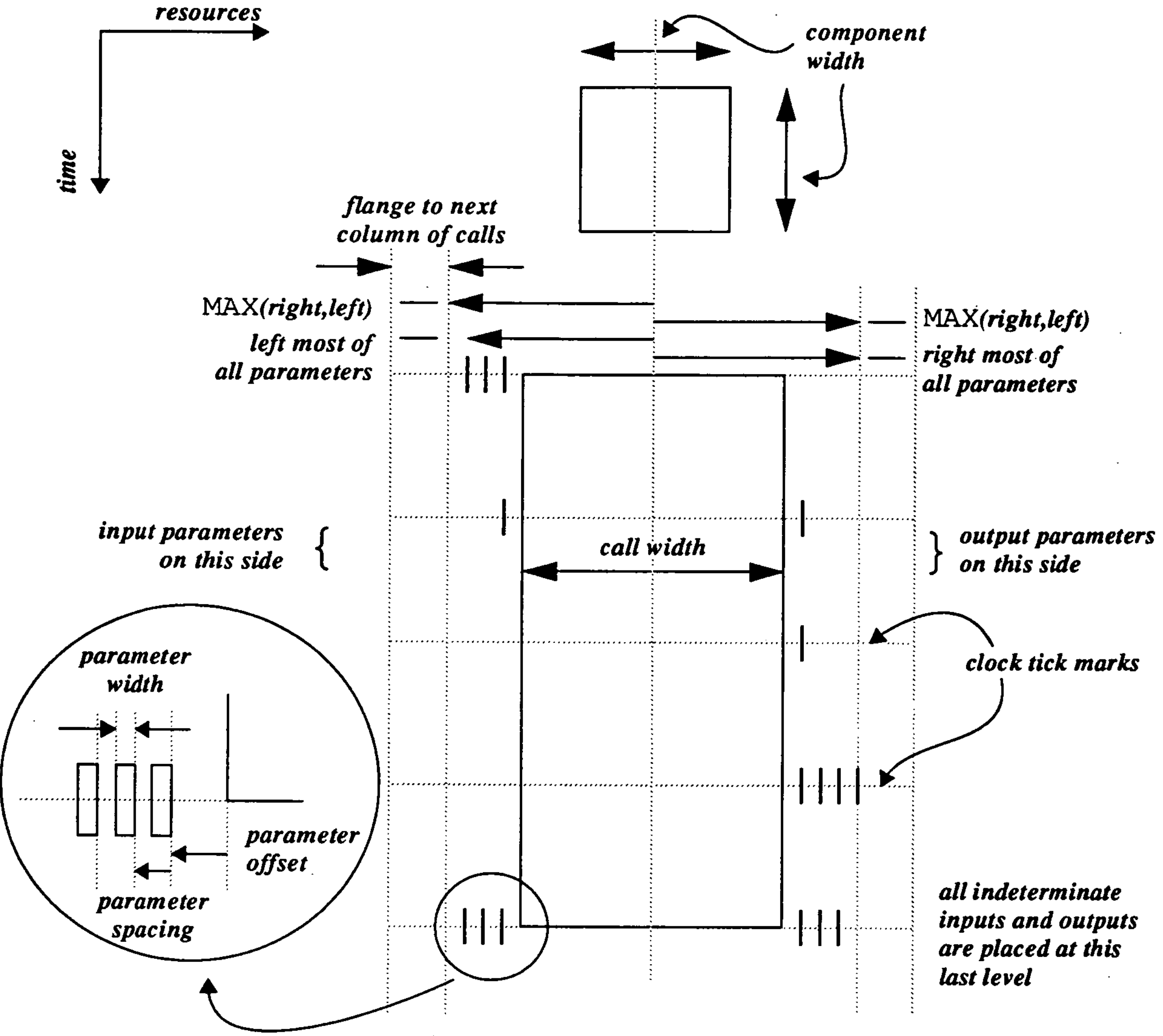


Figure 4-20. How a Call Looks Close Up

Having a well defined call shape, a definite parameter location, the next problem is related to how arcs passing from call class to the next call class, should actually be represented. The main problem arises because the definition of parameter is different if the call is combinatorial or clocked. Figure 4-21 illustrates the four cases of generation and consumption. The interesting point is that in all cases, the output is aligned with the start of a parameter, never with a clock tick. Whereas for clocked inputs, the alignment is with the clock tick.

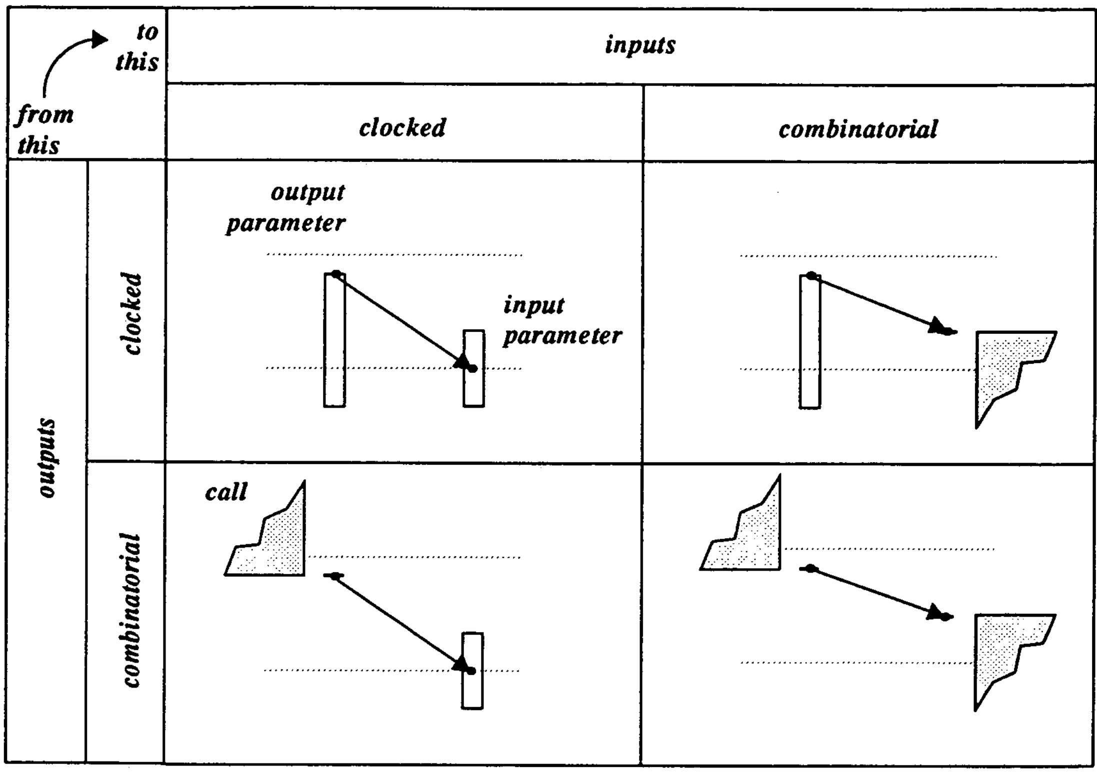


Figure 4-21. Arc Alignment, Depending on the Nature of the Call.

Data flow arcs are not the only form of arc that appear in resource-time graphs. By supporting the idea of making calls invisible, and if the associated arcs have not also been made invisible, they also need to be displayed. For such cases, the alignment is with the left or right side of the column within which the visible call that is effected exists. For arcs that have source and destination marked as invisible, then no arc is actually displayed. The other special case of arcs are constraints, which specify how one call must be placed in relation with another.

Calls are organised in columns, reflecting the shared resource or component that they actually operate. This is reflected by the presence of a column, at the head of which there is a component representation provided there is a resource allocated. With this interface, there are a number of design actions that can be identified, and these are explored as part of the design example shown in section 7.1 on page 176.

4.8.2.3 Mapping

One of the most novel concepts embodied in the SAGE data model, is that of mapping. In general, all calls associated with a given behaviour must work on the set of components associated with that same behaviour. On many occasions, an expensive component, (maybe in terms of area), might be present in the component list of another behaviour. From a user's point of view, rather than create another instance of that expensive component's structure, the better decision is to make use of this already existing component. The process of mapping provides a way to achieve this.

The process of mapping is complex, in that the already existing component is marked as having been mapped in the behaviour associated with the structure containing the said component. It is the using call of the behaviour associated with the structure containing the said component, which must instantiate the actual component and relate it back to the mapped (or formal) component. Since there can be multiple uses of the mapped component, the mapped component is treated as being utilised for the duration of the behaviour containing the mapped component. This simplifying approach meets the causal requirement that conflicting uses of the component are not allowed.

From a display point of view, the duration for one execution of the mapped component, and one execution of the behaviour of the structure that the component is in might be different. As a result, as well as the actual call to the behaviour containing the mapped component being represented, overlaid on top of this is a mapped call marker, in the form of an arrow depicting that the use of that component is made somewhere in it. The actual mapped component associated with a call, as opposed to the behaviour containing that component, is marked by a simple cross. These ideas are illustrated in figure 4-22

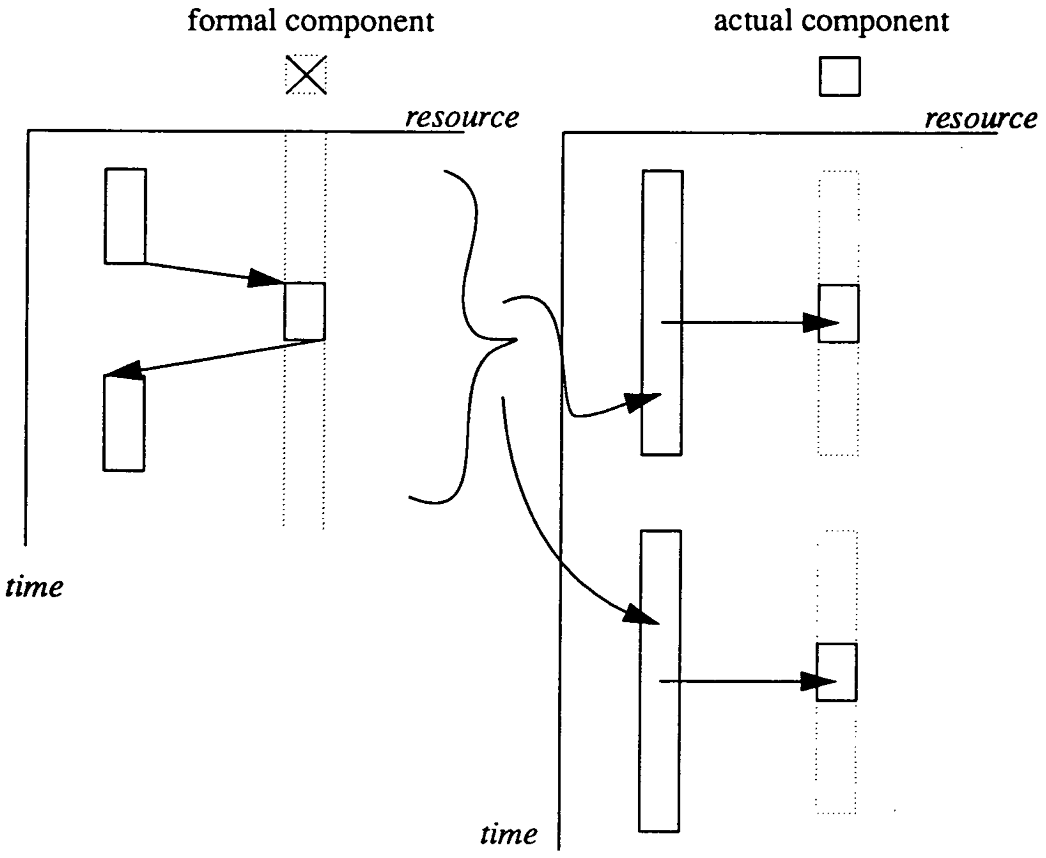


Figure 4-22. Mapping Example

4.8.3 Control Flow Graphs

The basic control flow in a behaviour is represented by a directed cyclic graph. The key features are related to disambiguating the node types. There are four basic types of node, reflecting the control structures that are extracted from the source code. There are fork nodes for branching, merge nodes for collecting control threads and fork+merge nodes for blocks that can exhibit both such activities. The fourth block class is that of straight line code, that does not have to be a collector or generator of multiple control signals. These different representations are illustrated in figure 4-23,

with the use of the flat edge on the icon to represent the collection or generation of control arcs.

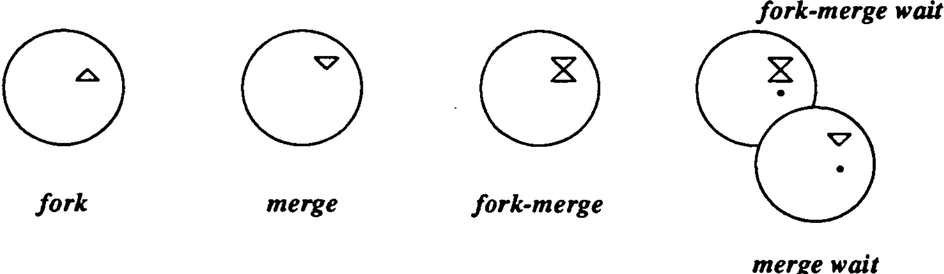


Figure 4-23. Control Node Categories

4.8.4 Structure Graphs

The structure graphs in SAGE, effectively have two forms of outlet, namely text display and schematics. From a designer's viewpoint, the schematic in general provides a much more useful medium with which to analyse a design. Much work has been done in recent years in trying to place and route schematic diagrams for maximum aesthetic appeal [59, 76]. Rather than adopt these same objectives, the requirements of having an orthogonally routed schematic displaying all relevant information in a timely manner was the driving consideration. As with all objectives of this nature, the problem was divided into a number of stages, of which the most important was routing and rendering. The process of producing the schematic, was greatly aided through the use of the creation approach described in section 3.2.1 on page 37.

With simplicity being the primary objective, this has led to the concept of the *multi-grid network* approach. The first stage involves finding one size for all components. With inputs arriving at the left of a component, and leaving from the right, the maximum height for a component can be determined from the component that receives the maximum number of inputs or outputs. Similarly, the maximum width of a component is determined by the number of bidirectionals present. Clearly, if there are no such pins, a minimum value is assigned. The grid that results, which is fixed for every component, is used to align pins. This same grid is used for wires that are

associated with the containing structure, through the assignment of virtual components that soak up the all the top-level pins. The next grid is that which contains the components. This is simply a n by m grid, where n and m are in general equal because of the simple row after row placement algorithm.

The routing channels are defined by this component grid, and unlike the first two grids, this implies a dynamic grid. In fact, the combination of all the components port grids, provides the routing grid. Whenever a pin needs to appear on a channel, then an exclusive route orthogonal to the pins direction is added to the grid. Thus the process of creating the rectilinear steiner tree [11] has been reduced to one of joining the connected parameters vertically, by creating a single vertical grid line, and then connecting the resulting wires horizontally on a nearest neighbour basis. This scheme lends itself to simple placement optimization and simple compacting of networks, the later achieved by simply folding nets in a channel onto each other when there is no implied electrical overlap. Figure 4-24 shows these three grids.

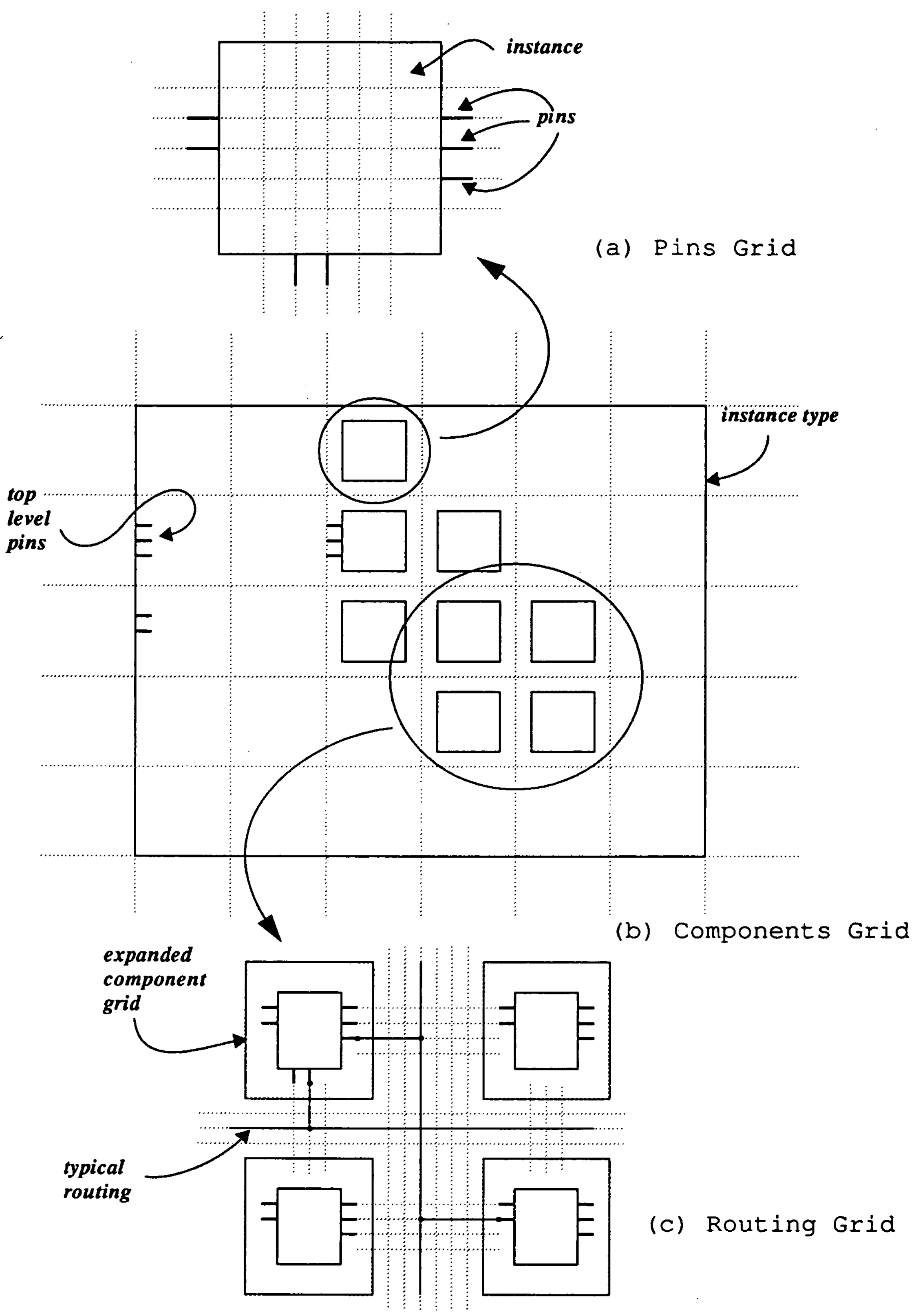


Figure 4-24. The Three Grids for Schematic Routing (Pins, Components and Routing)

As with the other graph classes, there are many aspects that can be displayed about a structure, the most significant being the nature of the component. In SAGE, components can represent one of six classes, namely: data processing, control, memory, address generation, constant generation or communications. Other issues relates to wires that connect to nothing, or that connect to only one pin. The former are not displayed, while the later are represented as blobs on a components pin. This disambiguates them from unconnected pins, which, following Mentor Graphics style, is represented with a diamond shape.

4.9 Interaction

There are several aspects that are reflected across all the previous visuals. The first relates to navigating around the design hierarchy. The second concerns interaction, both in a query sense and being able to modify a design to reflect a designer's requirements.

All the visuals are related in that objects highlighted in one graph, will automatically be highlighted in other graphs that might have the same object present. This happens in particular to all the key SAGE data model objects, namely, behaviours, structures, components, blocks and calls.

A typical design can represent complex hierarchies and interrelationships. To help a designer identify where they are in this hierarchy, it is useful to have a navigation aid that shows the path that has been followed. Figure 4-25 illustrates the navigation panels used in SAGE, and how they relate back to the SAGE data model objects. Whereas most navigation aids represent only one path, these panels have to illustrate

two paths, one for the functional aspects of the design unit, and the other for the structural.

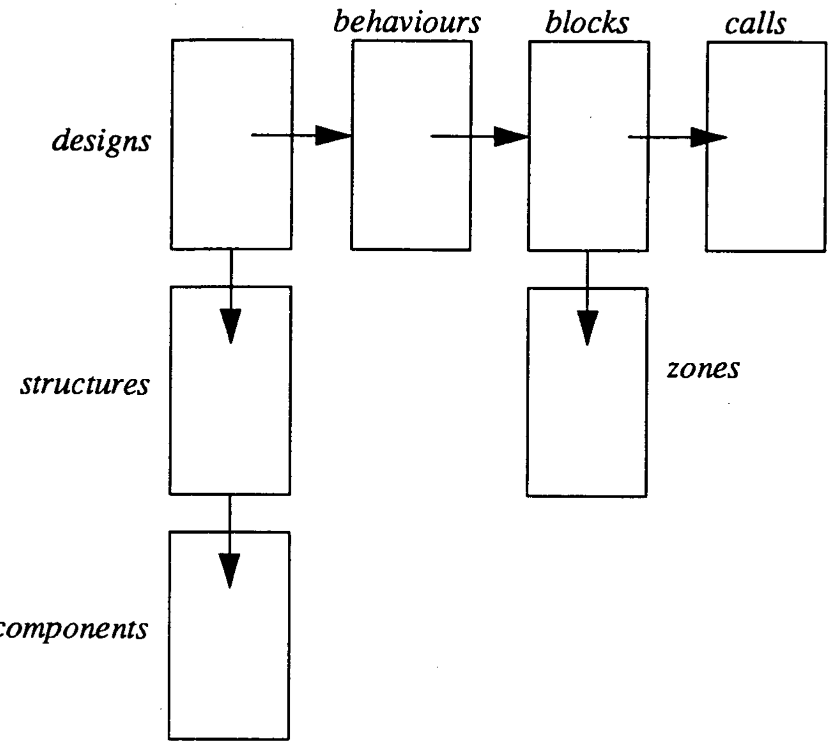


Figure 4-25. Navigation Panels

There are three key design actions that a user is interested in. There is the need to identify which structural objects can achieve the requirements of particular calls. Next there is the process of binding a given call to work on a particular component, whether or not that component already exists or is one of the structural objects returned by the match action. From a designer's viewpoint, this action saves on hardware and/or can associate real hardware with calls. If the target component is in another component and is mapped, then a structural mapping is implied, provided, of course, the component is capable of supporting the required calls function. The third action available to a designer is that of adding and removing constraints, so a particular schedule or execution pattern can be obtained.

As a set of operations, these are not usable without support in the form of simple copy and delete facilities, that handle general housekeeping functions. Another issue related to usability, is group selection. Without this form of feature, a user would be required to painstakingly iterate on a call by call basis for even a simple bind action.

To those accustomed to the precise, structured methods of conventional system development, exploratory development techniques may seem messy, inelegant, and unsatisfying. But it's a question of congruence: precision and flexibility may be just as disfunctional in novel, uncertain situations as sloppiness and vacillation are in familiar, well-defined ones. Those who admire the massive, rigid bone structures of dinosaurs should remember that jellyfish still enjoy their very secure ecological niche.

-- Beau Sheil, "Power Tools for Programmers"

5. Framework

With the complexity of problems that synthesis is aimed at - of the order of several hundred to tens of thousands of gate equivalent complexity, one of the most important technical problems is that of ensuring adequate software performance. This is even more so when interactive performance is required. Rather than adopt the philosophy of the software developed being simply experimental in nature and therefore the goal of performance need not apply, the opposite view was taken, which required many technical problems to be analysed and then suitably eliminated [98].

These problems appear at several levels, from subtle behind the scenes behaviour whose benefit would only be apparent when handling real world size problems, to directly impacting the way a user interacts with a system, This is not a hard and fast dividing line, but is an adequate categorisation for the flow of discussion to explore and describe the issues involved in the following sections. To begin with, the next section looks at the background decisions that directed the overall form of the framework elements.

5.1 Background Decisions

As with all projects, decisions are taken that can be considered immutable for simple reasons such as availability, cost and time, as well as the effect of inertia once such a decision has had significant impact on the development life cycle of a project. In the case of SAGE, several reasoned decisions were made, which formed a platform on which the SAGE development was able to happen.

In the case of the user interface, the `X Window System` [55, 56] was chosen to satisfy the design requirements of portability. Since the `X Window System`

provides only a raw low level graphics interface, OSF/Motif [57] was selected as the high-level interface to provide an industry standard look and feel. As there is a considerable standardisation effort within the IEEE on operating systems, POSIX was the selected target operating system. Since this is still being developed, UNIX BSD4.3 as supported on HP/APOLLO and SUN workstations formed the development environments actually used. The decision to use ADA [3] as the main programming language, is always controversial. Nevertheless, after careful consideration, it was quite clear that ADA has many excellent language features that are essential for large, multi-purpose projects, and has a style that lends itself to good software engineering and easy documentation. The choice of ADA raised the problem of interfacing to Motif and X. As a vehicle to achieve this interfacing, it was necessary to use the native language of the operating system environment, namely C [8].

Within these boundary conditions, it was necessary to develop concepts that could solve the problems of combining X, Motif and ADA in a way that would satisfy the objective of providing high-performance interactive portable window facilities.

5.2 Process Model

A careful look at the small print in the X Window System, highlights one interesting fact. Once an application has a handle on a server object, regardless whether or not that application created it, the application has full rights to that object, such that it can treat the object as if itself did create it. Most applications assume that they have complete control over their output device, with X they don't, (and in fact this is a very serious security loophole). Nevertheless, without this feature, the following described SAGE process model would not be possible [25].

Figure 5-1 shows the overall SAGE process model in a highly stylised form. There are shown three major software modules with interconnecting lines depicting the information flow required. Each module has an icon depicting the broad class of activity that the module provides. The ADA module, forming the bulk of the SAGE system, handles all the synthesis activity. The X module is responsible for all drawing

(it is effectively the server), while the Motif module provides the user interface facilities. From the ADA modules perspective, it receives requests from the Motif module, each of which it obeys as a series of actions. One of the actions could be the requirement to render a graph, which it does by communicating the graph to the X module. Once it has completed such a request, it communicates back to the Motif module, that the design request is complete. At any time during a request, the ADA module can communicate informational messages to the Motif module. Because of the nature of information carried, the bandwidth requirements of the Motif-ADA modules link are generally much less, by orders of magnitude, than that between the ADA and X modules. Consequently, the gain in defining a human readable protocol between Motif and ADA modules, as opposed to a compact binary representation, far outweighs the slight loss in performance.

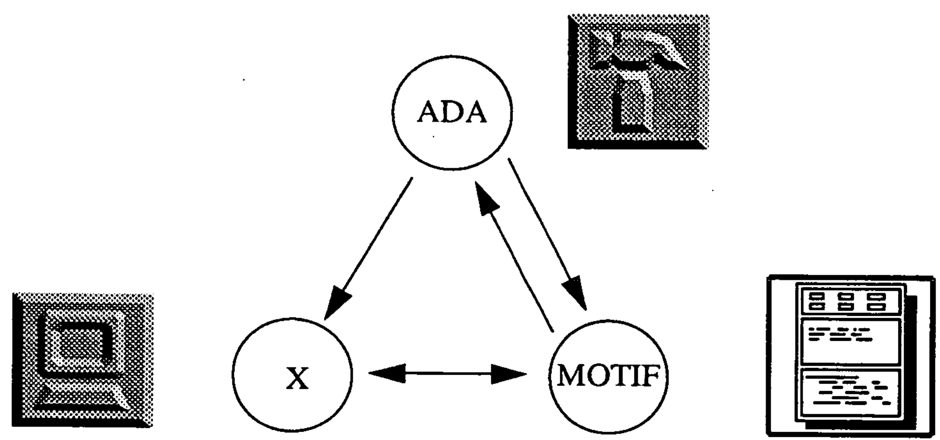


Figure 5-1. Stylised* SAGE *Process Model

Each of the modules exists within its own process space. The X module and Motif modules are both event driven, while the ADA module has only a single thread of control. In relative terms, it is important to restate that the communication between X and ADA is required to handle a much higher capacity than the link between Motif and ADA, and therefore does use a compact binary level representation.

Although this process model has been presented as a *fait accompli*, it is nevertheless interesting to examine the rationale behind the resulting process model. Since the majority of data-structures are held within the ADA language environment, it was

necessary to generate draw commands directly from this database in order to maximise performance. This led to the requirement for an ADA language binding to X, that needed to cover only the basic drawing commands. Motif on the other hand, is based extensively on the X Toolkit and so makes extensive use of callbacks. Since ADA provides no formal means of letting foreign languages call its subprograms and hence has no callback facility, the natural solution was for Motif to exist in its own process and communicate with the ADA module through the use of UNIX pipes. Trying to force the Motif and ADA modules into the same process address space would achieve little, since the event driven nature of the Motif module, would require that it exist within its own ADA task.

This division of drawing responsibility between Motif and ADA is illustrated in figure 5-2, where the *exact* destination of the arrows on the display surface illustrates how ADA handles what happens within a window, while Motif handles the general issues involved in managing that window. Other important features of the process model are also illustrated by this model.

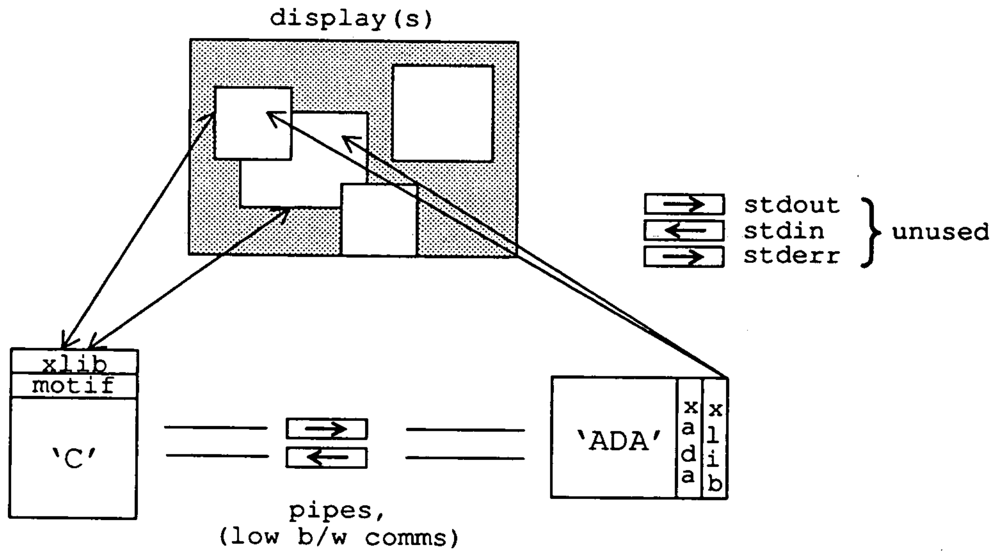


Figure 5-2. Process Model - Normal Operation

Note how the ADA process does not use the three standard UNIX pipes, commonly referred to as *stdout*, *stdin* and *stderr*, but instead has its own pipes. (At job creation

time, the Motif process informs the ADA process through command line arguments as to the location of these pipes in the file descriptor table). Two major benefits arise through this approach.

Firstly, subprograms in the ADA process can use *stdin* and *stdout* during the debug cycle. Early versions of SAGE, which used these channels for communication, were very difficult to debug because any information had to therefore be directed to a file, and this file could not be viewed until the end of a SAGE session. Even if another version of ADA existed which supported 'many readers/one writer' on this file, because no input is allowed, the only way to affect the flow of control is to use a debugger. When a debugger does need to be used, it generally has to be applied directly to the ADA process. Similarly, when testing, only the ADA process is required to be invoked. The ADA process can support this simple requirement by folding its main communication channels onto *stdin* and *stdout* by recognising whenever it is invoked without command line arguments. Note, this arrangement still means that such an independently invoked ADA process can still draw graphics, since all it requires is a named window on a given screen. This is illustrated in figure 5-3. Clearly, the disadvantage is that when doing graphics debugging, the programmer has to set up and manage 'dummy' windows, so that the ADA process can drive a named window. In practice this is straightforward, since the X Window System 'xwininfo' command can be used to return a handle on any window that can be pointed at, and this can be manually passed to the ADA process.

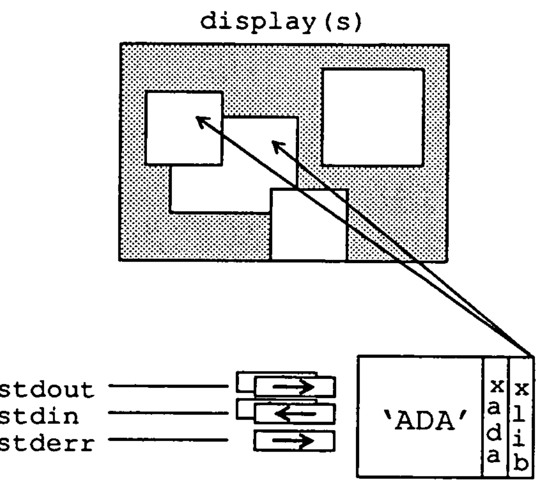


Figure 5-3. Process Model - Debug/Test Operation

The second benefit relates to the way the Motif-ADA communication pipes can be actually used. As stated earlier, for simplifying reasons a human readable protocol has been used, but it can easily be imagined how this could be tokenised for a first level gain in efficiency. The next area concerning improved efficiency then relates to how UNIX pipes are utilised. By having the creation and control of the main communication pipes directly within the scope of the ADA and Motif modules, then there is no need for a line buffering shell which is wrapped around stdin and stdout in a normal ADA process. In fact, this approach is mandatory if no line break characters are required as part of the protocol - as is the case. Instead, as would be natural, the protocol itself implies when a message completes. With the LISP [10] style protocol, this simply means a matching closing bracket - ')', to the starting bracket - '('. Another aspect of managing these pipes relates to the event driven nature of the Motif process. This means that when the Motif process is sending data it must not lose its thread of control to the ADA process by blocking on the pipe if the ADA process is too busy to actually receive the data. Similarly, only when the ADA process has data to send can the Motif process read its input pipe.

On the minus side, there are two problems that arise. Since there are two processes that need graphics support in the form of X Lib support, it may appear that the memory space occupied by the Motif and ADA processes might be twice what it

need be. In fact, because of the recent support of shared library facilities in UNIX, the memory penalty is incurred for only duplicating state variables -which is in general of the order of tens of kilobytes at the most. The second problem relates to race conditions that can arise when there are significant delays associated with the three stages.

Although this process model as described is used for SAGE, it has been designed to handle five other requirements.

• Firstly, the need for undoing and redoing design actions. Rather than being myopic and expecting the application itself to handle this function, by taking a step back and recognising how easy it is to replicate UNIX processes in a time and space efficient manner (relying on virtual memory management systems that use copy-on-write), then this is just a step away from fast, efficient and reliable multi-level undo/redo facilities. The change to the process model is one additional pair of communication channels for each additional copy of the ADA process. For a single level undo/redo, two ADA processes are always present. One of the ADA processes is managed as the primary process, while the other process represents the design state before the last design action. An undo/redo activity would simply toggle between the processes, while a new design action that modifies the design state would require the older ADA process to be replaced by the current ADA process, while a copy of the current ADA

process has the design action applied to it. This is illustrated graphically for a single level undo/redo facility in Figure 5-4.

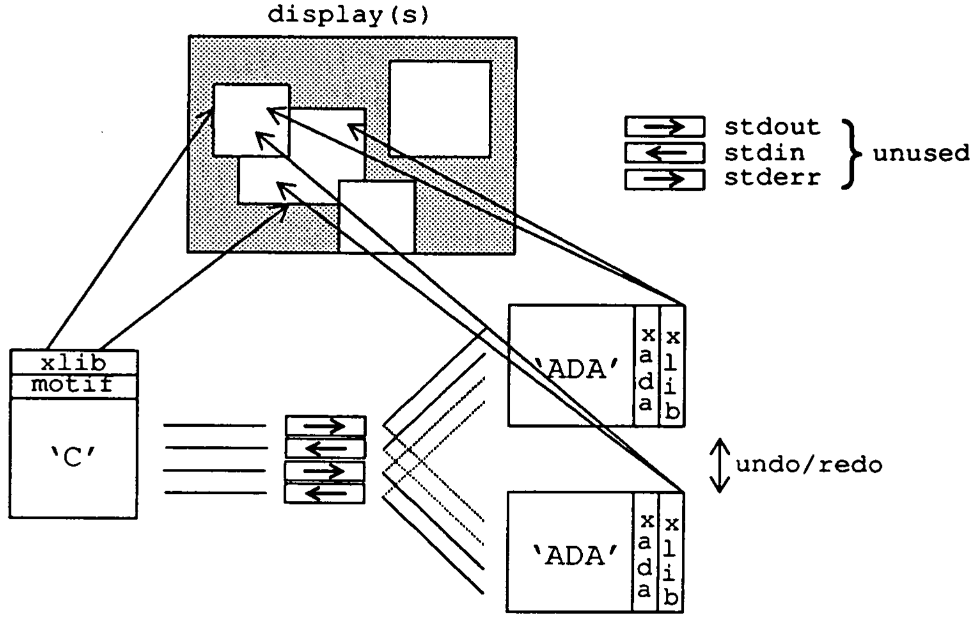


Figure 5-4. Process Model - Dual Process Operation

• The second requirement is that of error resistant behaviour of the SAGE system. With complex prototype software, system crashes are the norm rather than the exception. Using the same technique as for single level undo/redo, any system crashes can be captured and presented to the user with the option of regressing back to the stage before the last design action. Because of the smallness and reliability of the X and Motif processes, they are assumed to be (and are in practice), much more robust than the ADA processes, and therefore any crashes in those parts would generally be unrecoverable.

• The third requirement can be simply stated as the ability to interrupt a design action and ensure that design state is returned to a consistent state. Although simply stated, the complexity of implementation needed for a single process model, generally means this feature is not supported or if it is, it is implemented in a fashion that merely terminates the interrupted process in an orderly manner. With the dual ADA process model, this problem is easily solved. The effectiveness of this approach cannot be understated, especially considering the opaque nature of many synthesis algorithms

which can claim the thread of control in an ADA processes for many minutes if not hours. It is possible to envisage that such algorithms could be modified to provide suitable interrupt points, but with prototype software the effort is directed at the development of the synthesis algorithms rather than supporting how a user might want to interrupt such an algorithm. For algorithms that do take hours (such as routing), having suitable interrupt points is essential.

• As mentioned, synthesis algorithms can take significant time to complete, and in general this would normally be the time that a designer would go off and make a cup of tea since the tool is in a busy state. The fourth requirement is to provide some subset of commands that will let a designer still continue to explore a design. In the dual process model, this would mean that while the latest process is actioning the design action, the last process is available for providing information about the design, whether that involves updating existing visuals or providing information in other forms. A more elaborate approach, which this process framework can support, is allowing the full set of commands. Although much more complex issues arise as to how the ADA processes manage their allocated windows, it is a sound mechanism to explore several design avenues in parallel.

• Fifthly, with processing power becoming cheaper, it becomes natural to divide systems into processes that can be farmed of onto separate actual processing units. As is the case with the SAGE process model, there are three processes with which the simple conversion of the Motif-ADA communications channels from pipes to UNIX IPC links can easily be managed on three separate machines. An early mock-up version of SAGE demonstrated this in action [25]. Note, although the multiple ADA processes have to reside on the same hardware because of the way UNIX works, they can still gain from parallelism by being executed on multi-processor UNIX machines. Since the support for parallelism might require IPC links, this also partly drives and justifies the need for communication channels separate from *stdin*, *stdout* and *stderr*. The reason is that IPC links are prone to several problems, not least of which is that there can be no guarantee that the number of bytes requested to be read or written from/to a channel are the actual number of bytes read or written.

Most of these five requirements are incremental on the basic process model that has been used for SAGE. The other key aspect is the transparency of these concepts in practice. This applies both to a user of such a system, and the programmer developing the algorithms used in the ADA module.

The final comment about this process model is that though it has been developed to solve the primary problem of interfacing X, Motif and ADA, it is in fact general purpose and could be used in many other application areas where it is desired that the user has to be maximally empowered, while minimising the programming task that is needed to achieve this.

5.3 Language Bindings

Formally, the X Window System is just a line protocol [54], comparable in style to the opcode/operand structures found in assembly code. In practice, this is only the first stage between several layers of abstraction that separates it from application code as illustrated in figure 5-5. Since all these interfaces were developed in the C world, they are immediately available to C and C++ [9] workers. With a language other than these two, there is a natural barrier that has to be circumvented by the development of a suitable language binding. Note, as far as and including the X Toolkit Intrinsics, these are internationally accepted standards, while the remaining layers represent different approaches of handling the look and feel issues.

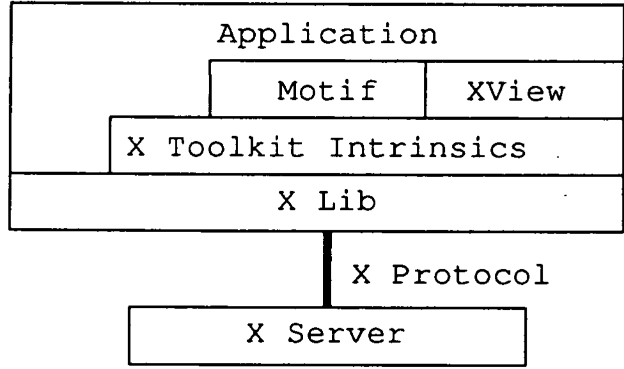


Figure 5-5. X Window System *Conceptual Layers*

A significant amount of work in the US has been invested in developing ADA bindings to the X Window System. Most notable, is work done by SAIC (Science Applications International Corp.), which has developed complete bindings to X11R2. This consists of over 25,000 lines of ADA code, plus some 16,000 lines of HP X-RAY toolkit bindings. Work now undergoing at SAIC, includes updating these ADA bindings to support X11R3/4, and work on ADA bindings to the X Toolkit Intrinsics as well as the Athena widget set. Rational has also developed a binding, but which works at the X Protocol level, rather than the X Lib level as in the case of the SAIC work. Like SAIC, if they have not done so already, they intend to make their work publicly available.

SAGE, in sharp contrast, has an ADA binding to X Lib that focuses only on commands related to drawing activities, the most involved aspects of which concern the handling of graphic contexts. This binding is less than 1000 lines long and works with X11R4. SAGE also has bindings to parts of the X Toolkit Intrinsics, the reasons for which are described in section 5.5 on page 139 on picture attribute management.

The mechanism that ADA provides for such language bindings, is a well defined part of the language, rather than a vendor specific construct. ADA supports foreign languages through the use of the 'pragma INTERFACE' construct. In relative terms, the solution to the name binding is simple compared with that of mapping types between different languages. Provided assumptions are made about the bit pattern layout of C types, ADA supports representation clauses to match to the externally imposed storage layout (this is the same concept as C's support for bit fields in structures). Figure 5-6 illustrates an example of such a binding. It also shows the use of representation clauses.

```
type XColor_Rec_t is record
   pixel : U_LONG;
   red, green, blue : U_SHORT;
   flags : CHAR;
   pad : CHAR;
end record;

type XColor_t is access XColor_Rec_t;
```

```
for XColor_Rec_t use record
   pixel      at 0 range 0 .. 31;
   red        at 1 * 4 range 0 .. 15;
   green      at 1 * 4 range 16 .. 31;
   blue       at 2 * 4 range 0 .. 15;
   flags      at 2 * 4 range 16 .. 23;
   pad        at 2 * 4 range 24 .. 31;
end record;


function XAllocNamedColor(
   display : Display_ptr_t;
   cmap : Colormap_t;
   colorname : STRING_PTR;
   colorcell_def : XColor_t;
   rgb_def_def : XColor_t
) return Status_t;


pragma INTERFACE (C, XAllocNamedColor);
pragma INTERFACE_NAME
   (XAllocNamedColor, "XAllocNamedColor");
```

Figure 5-6. PRAGMA and REPRESENTATION Clause Example

The publicly available SAIC X-ADA bindings were not used for reasons of poor support, reliability, quality and most important lack of maintainability. Its major failing is the inclusion of an additional calling layer, between a users draw request, and the final call to the interfaced language. In principle this can be inlined out of existence, but in practice, the extra call level includes additional computation such that the process of inlining can be computationally very expensive. Other problems range from simple bugs with basic commands like circle drawing, to the omission of bindings to the X Macros (which are provided in function form within X Lib).

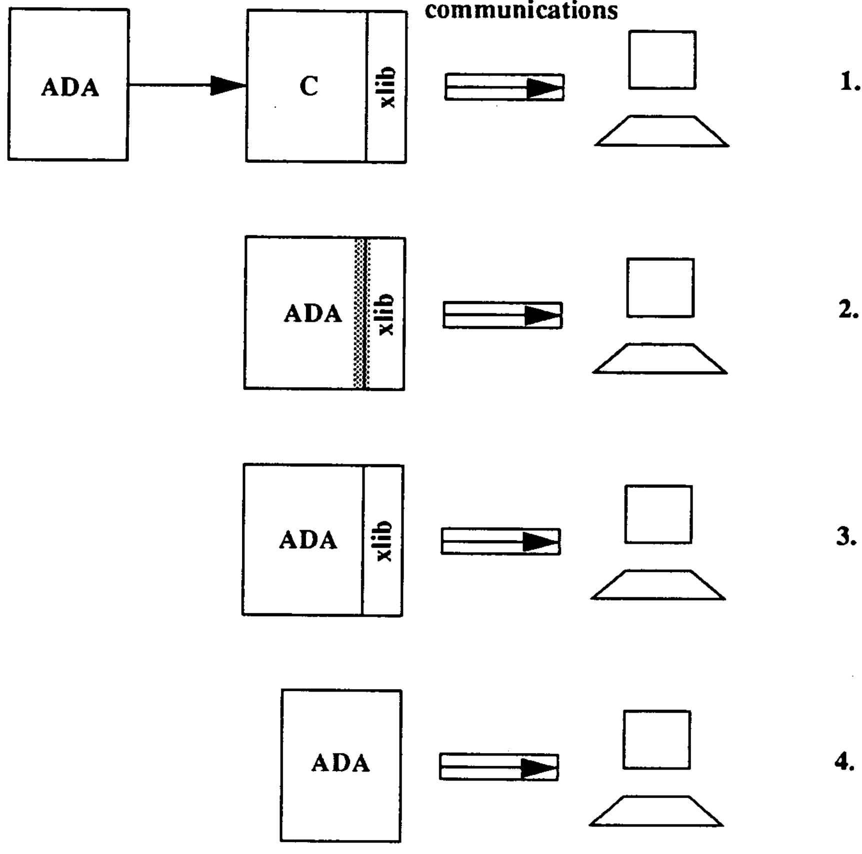


Figure 5-7. X-ADA ***Binding Scenarios***

Figure 5-7 shows the various levels of language bindings that are possible, and where SAGE, SAIC and Rational bindings are placed. The first diagram is the form that was adopted in an early version of SAGE (version 2). The ADA process would construct each draw request into a textual equivalent and transfer it across a UNIX pipe to some C code that would parse the request and pass the request onto X Lib. The second diagram shows the structure imposed by the SAIC binding, with the stippling indicating the presence of an extra procedural level. The third diagram is the language binding adopted by the latest version of SAGE (version 4). It is interesting to note, that if we replace the word ADA, by C, we would have the exact form of client structure found in standard X Window System applications like *xclock* and *xterm*. The fourth diagram, illustrates the form of the Rational binding. Here, the binding is direct to UNIX IPC channels and as a consequence appears to be relatively small. This in

some ways is the ideal binding, since the ADA programmer is now provided with a completely rationalised interface to X, free of C code. Nevertheless, there is the need to replace X Lib. This is the failing of this approach, since not only is there the problem of ensuring faithful replication of the X Lib behaviour - not easy for an interface measured in tens of thousands of lines of code, but there is also the problem of tracking the improvements in efficiency and scope associated with the definitive X Lib definition in C. Rather than indulge in the polemics involved in the ADA and C debate, the pragmatic observation is one that recognises that there will always be many more C programmers using the X Window System than ADA programmers, consequently, the third option is the optimal.

5.4 Drawing Model

There any many systems that can provide object level display capabilities and many issues in this area of computer graphics have in general been resolved. The common presence of such software in action from the simplest drawing package to the latest CAD packages hides the fact that there are many ways to achieve what appears to be superficially the same end result. With this variety comes a lack of direction in which display problems are actually being solved, and as a consequence there are many defects present in the rendering models used in many current graphics packages. These problems are reflected in simple details like, for example, not supporting incremental update, or if it is supported, missing out details like updating what was behind any deleted object. What appear to be simple problems hide the complexities of the implementation needed to solve the problems.

To illustrate the nature of some of the problems that are present, consider the phenomenon of 'screen droppings'. This is a good example of where it could be argued to be a system feature, or in fact a minor bug. The solution could be simple - simply update the entire screen after each action that might leave screen droppings. But this is irritating to the user because of the flashing behaviour it can result in. Another solution might be to draw an off-screen picture and copy it back to the display. This is expensive primarily in memory resource. Yet another solution might be to only support two colours, a and b, and use the technique of exclusive-or'ing, of

only ever drawing in $(a \oplus \bar{b})$ or its inverse. But this again has a problem, namely of having a graphical interface that supports only two colours. Problems of this nature and the difficulty or cost of the solution, mean that many rendering methodologies are usually a selection of the easiest solution - and for this example it means the user has to just put up with the screen droppings. Where the rendering model for SAGE is different, is that *no* such compromises have been allowed in its design and development.

The need for providing an optimal rendering model is doubly important in the presence of the X Window System. As a concept, the X Window System supports the ideas of client/server processing excellently, but at the cost of increased processing compared with what native graphics drivers can provide. Thus, even with a language level binding that maximises performance by binding direct to X Lib language calls, care therefore has to be taken when actually drawing.

In particular, the SAGE rendering model addresses the needs of accurately drawing layered 2 dimensional graphs with:

- fast and efficient region query,
- fast and efficient point query,
- fast and efficient region repair,
- multiple viewports,
- detail culling,
- size invariant object handling (such as text) and
- composite objects.

Additionally the model batches drawing actions in order to minimise server communications and so improve overall throughput. Although not discussed, but shown in figure 3-8 on page 37, the model can easily be directed to produce output for other display technologies such as PostScript and HPGL. Figure 5-8, shows the broad outline of the rendering model used in SAGE.

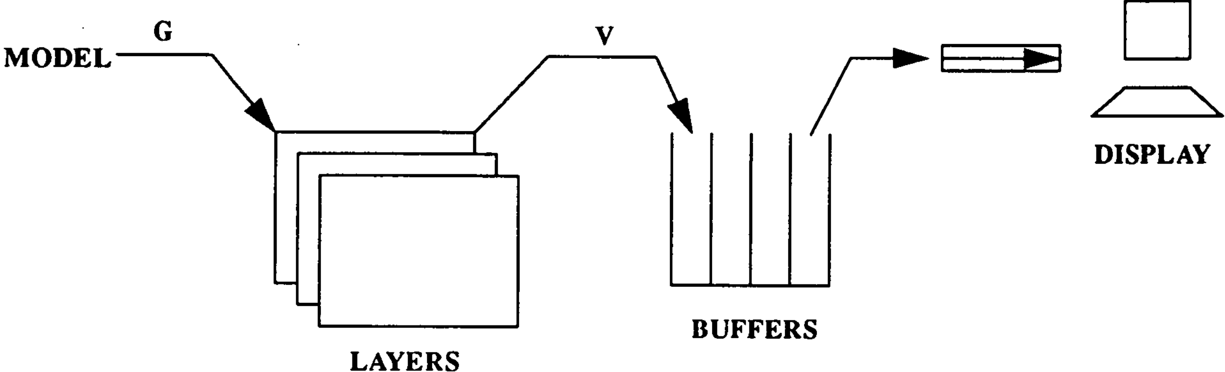


Figure 5-8. Rendering Model Overview

The range of basic objects that can be drawn include lines, rectangles, polygons, arcs, marker shapes, text and objects hierarchically composed of these objects, (or, as referred to earlier: composite objects). All these objects are represented by their basic bounding rectangle in a quad-tree based data-structure as described in section 6.3 on page 162, that supports the fast and efficient region query, point query and region repair requirements.

Objects placed on one of the layers, pass through a matrix operation called `G` (for generation) when the spatial data-structure is built, as well as through the matrix `V` (for viewing), when the object is viewed. Clearly, for performance reasons, the matrix transformation is in practice applied only once at the point of use, as in `V*G`. This same approach to leaving the matrix operations to as late as possible, also applies in the case of composite objects as illustrated in figure 5-9. When instantiating a composite object, there is a second matrix called `G2` (for generation again) which provides an opportunity to scale the composite object, and a matrix `T` (for translation), which effectively contains the `(x, y)` coordinates of where the origin of the composite object should be placed. Thus when the object database as shown at the second level in the figure, (b), is viewed, the viewing matrix for a line in that picture would be `G*G2*T*G*V` (where the two `G` matrixes need not necessarily be the same). As shown by the third level, in the figure, (c), the extension to further hierarchy in composite objects is straightforward.

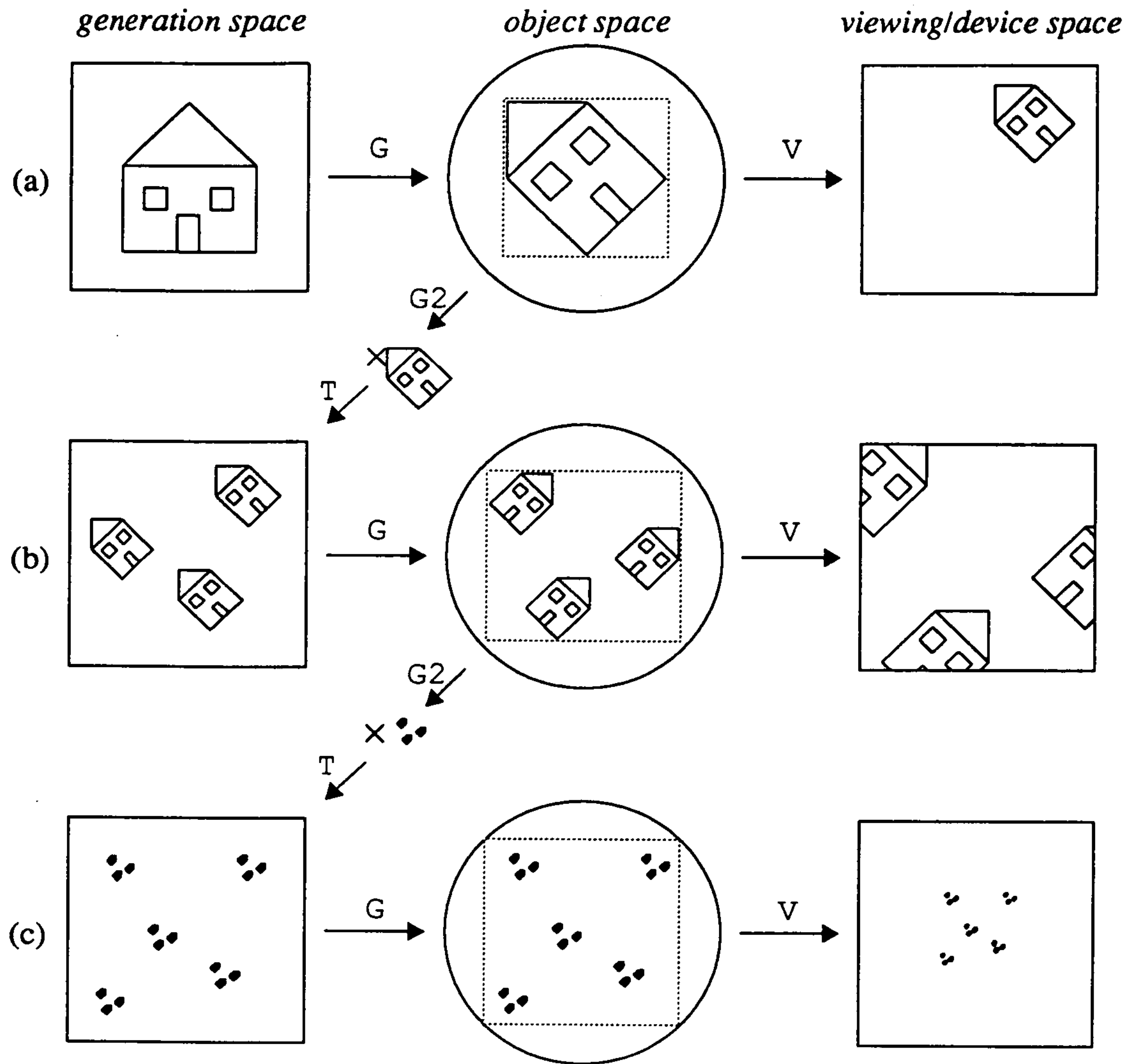


Figure 5-9. Composite Objects Through Hierarchical Construction

Now, consider the problem of a query or repair operation. The region to be handled could be specified in device (or viewing) space coordinates or in object database coordinates. For the former, a reverse application of V is required, namely V^{-1}. The only thing to watch out for, is that in the reverse transform, a rectangle region might be transformed into a non-rectangular shape in the object space, in which case this skewed rectangle must be orthogonalised, in order to ensure that we are searching as much of the object space as is needed by the target device space because only rectangular shaped queries are supported by the quad-tree data structures used in the

layers. When the region is specified in object space, a similar activity has to happen, but twice over. Clip regions in the `X Window System` are rectangular, and as a result, if the mapping from the object space produces a skewed rectangle in device space, it has to be orthogonalised and then mapped back to object space and orthogonalised again to provide the final region to actually traverse. This ensures that the device space region that is affected is fully covered, which is especially important for region repair requests. (Obviously the fact that the matrix G has not been applied causes the method to be slightly more complex in practice than as actually described, but the principle is the same). Figure 5-10 illustrates graphically this mapping process. Note, the process of orthogonalisation is defined simply as the maximum enclosing rectangle aligned to the axis of the device or object space axis.

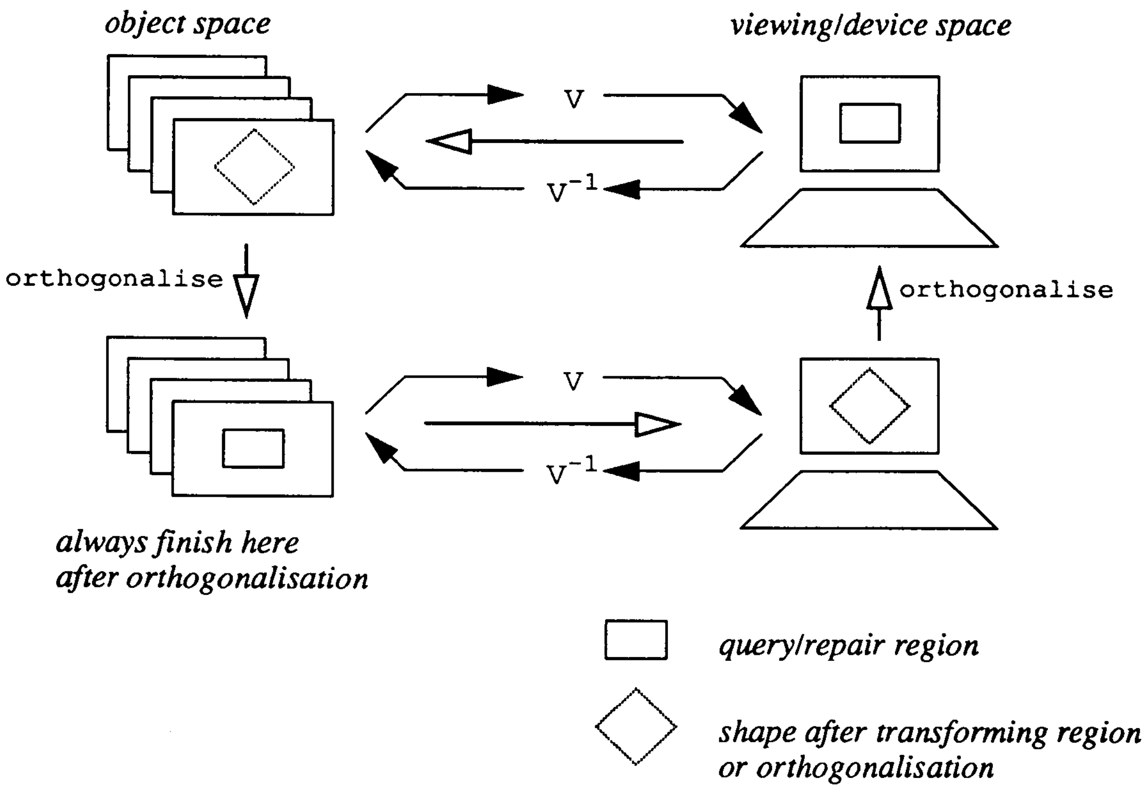


Figure 5-10. Region Mapping in Device and Object Spaces, for Non Orthogonal Mappings

By having layers, two display problems are overcome. First concerns the stacking order of any objects to be drawn. The layer abstraction means that the components of a picture can be managed much more easily, even to the extent that visibility of

objects can be controlled by simply swapping layers. Secondly, overlap within a layer can be left as an issue for the user of the abstraction, for example, a layer containing lines crossing will generally cause no problems. An additional benefit is that of helping incremental update. For example, if a new object is added to the top layer, than only the region specified by its bounding box in the top layer needs to be traversed - which provides for fast and efficient highlight activities.

Most rendering models do not take into account the real thickness of objects drawn, compared with their mathematically thin representation in the object database, nor the fact that sometimes the objects size are scale invariant (such as text or marker shapes). The subtle screen errors that could result if these problems were not addressed are avoided in the SAGE rendering model. This happens by always ensuring that a given region is first extended to reflect how in reality the objects might extend beyond their bounding boxes. This region forms the target clip space. This region again has to be extended for objects that might straddle it, but from the outside; but this time, the region specifies the search space. Figure 5-11 illustrates these three regions and how they lead to reliable drawing of thick lines.

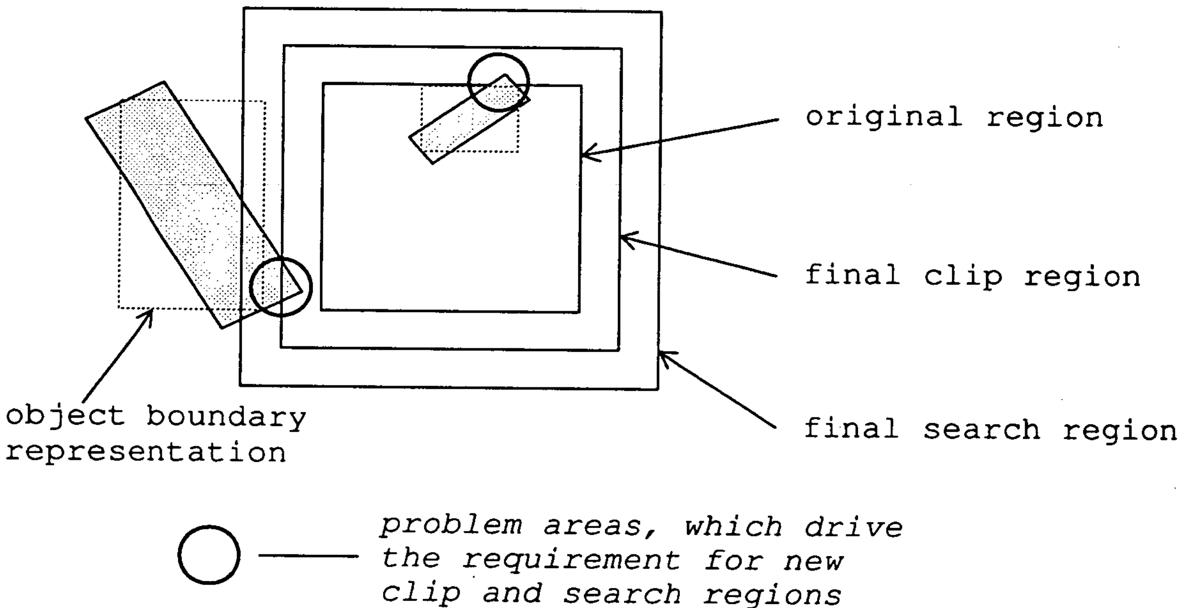


Figure 5-11. Clip and Search Regions

This rendering model handles the two most common problems with point queries, namely not being able to get close enough to the object to select it, or always selecting the wrong object. This is achieved by the point query in fact being a region query sized in device space of a few pixels square, so as to provide a greater chance of selecting the object pointed at by a user. As with the region query, in fact a list of all the objects encountered on selected layers is returned, ordered by a distance measure between each object and the query point. (This can still lead to problems if the fidelity of the distance measure is not accurate). This approach also ensures that if there are multiple objects at exactly the same point, no arbitrary decision is taken to return one of them. Minor issues like only measuring the distance from the perpendiculars of the border of an empty rectangle are also handled. A final point relating to the query operation, is that, like with schematics, data can be associated with database objects. Consequently there may be many objects with no data, or with the same data and therefore these redundant objects can be automatically filtered during the query operation to significantly improve performance.

When viewing part of a graph, as much batching of draw requests is made as possible, since this provides dramatic improvements in `X` server behaviour. Draw requests are binned according to the sort of atomic draw request required, and what attributes (discussed in the next section) are being used. It is interesting to note that this makes multi-draw commands like `XDrawLines` and small bitblt pixmap operations very inefficient compared to the use of `XDrawSegments` and `XDrawPoints`. As mentioned already, significant extra performance is obtained by the fact that the model also supports culling in region repair, but since this is a result of the quad-tree data structures used it is discussed in section 6.3 on page 162.

5.5 Attribute Management

The six basic atomic objects that can be drawn (lines, rectangles, polygons, arcs, marker shapes and text), have, as can be imagined various attributes that can help distinguish elements within the same atomic object class, as well as against each other object class. There are in fact, a total of eleven possible attributes, which, for efficiency reasons, correspond very closely to the `X Graphic Context` attributes.

As well as straight forward foreground/background colouring attributes, there are dashing attributes and various fill attributes based on tiles and stipples. The variety of these attributes reflects the need to differentiate objects displayed on workstations that provide monochrome, grey scale or full colour displays (i.e. eight planes or better).

As a general point of philosophy, the X Window System encourages application developers to make their systems as customisable as possible by the end user. With the variety of objects that are displayed through visualisation in the SAGE system, it was clear that the application of this philosophy was essential rather than just desirable. Rather than re-invent the X Resource Manager within the ADA environment, a handful of ADA bindings were made to the relevant resource manager facilities within X Lib and the X Toolkit Intrinsics. (Note, the use of this facility is distinct from the use made by Motif of these same X Resource Manager facilities).

Whereas the normal resource manager specifications tend to follow a widget hierarchy as imposed by the X Toolkit, no such constraints were present in developing a suitable class name for use within the ADA hierarchy. This added freedom has allowed the provision of three very useful facilities when specifying attributes for the ADA system.

• Firstly a user can positively discriminate between attributes for colour workstations, and those for black and white workstations. Despite the wide variety of visual[†] classes that the X Window System supports, the broad distinction between colour and mono, especially for most standard workstations that SAGE has been targeted at, is a sound division.

• The second facility, is an ability to discriminate between different workstations, by having in-built in the class name a *machine /server/screen* tuple. By selectively choosing attribute values for different workstations, the same graph can be viewed on two separate workstations, highlighting different aspects of the same graph.

†. 'Visual' in the sense used in X documentation.

• Thirdly, the actual attribute can be hierarchical. This allows the full power of the resource managers wild carding facility easily to specify values for the majority of cases, while being able to particularise specific instants. In figure 5-12 is an example of the full class name that can be matched to, as well as an example of how an attribute is specified in ADA with its corresponding usage within an X defaults file.

- *example name:*

```
sage.monroe.0.0.colour.att1.att2.foreground
```

- *corresponding class:*

```
Sage.Hostname.Server.Screen.Colour.Attribute.SubAttribute.Foreground
```

- *general class structure:*

```
<s/w>.<display tuple>.<colour>.<hierarchy of attributes>.<type of value>
```

- *ADA attribute definition:*

```
NG_COMP_DATAPROCESSING_INTERNAL =>
   GET_ATT(
      ATTRIBUTE => "ng.comp.dataprocessing.internal",
      DEPTH => 4,
      CLASSES => (
         Foreground_c | Tile_c | FillStyle_c => TRUE,
         others => FALSE
      )
   ),
```

- *corresponding X defaults entry:*

```
Sage.*.colour.ng.comp.*.internal.foreground: lightblue
```

Figure 5-12. Picture Attributes Example

The actual attribute value (the last part of the class name), takes on the usual values associated with a graphics context - features such as linewidths, background pixmap etc. One exception is the use of an attribute called 'CleanText'. This attribute is not available as part of a `X Graphics Context`. In use, it determines if text calls can clear their background before display of the text, with the trade-off being between improved legibility or fully visible graph information.

These features mean that only one application defaults file needs to be maintained for SAGE to work on a wide variety of X based workstations. In particular for hosts making use of default attributes, dynamic selection of colour or monochrome attributes can be made by SAGE. A naive implementation would simply allocate all the resources needed for all the attributes encountered on a per *machine/server/screen*

basis. This is inadequate, since not only is there overhead in allocating resources that may never be used, but duplicate resources could be easily allocated.

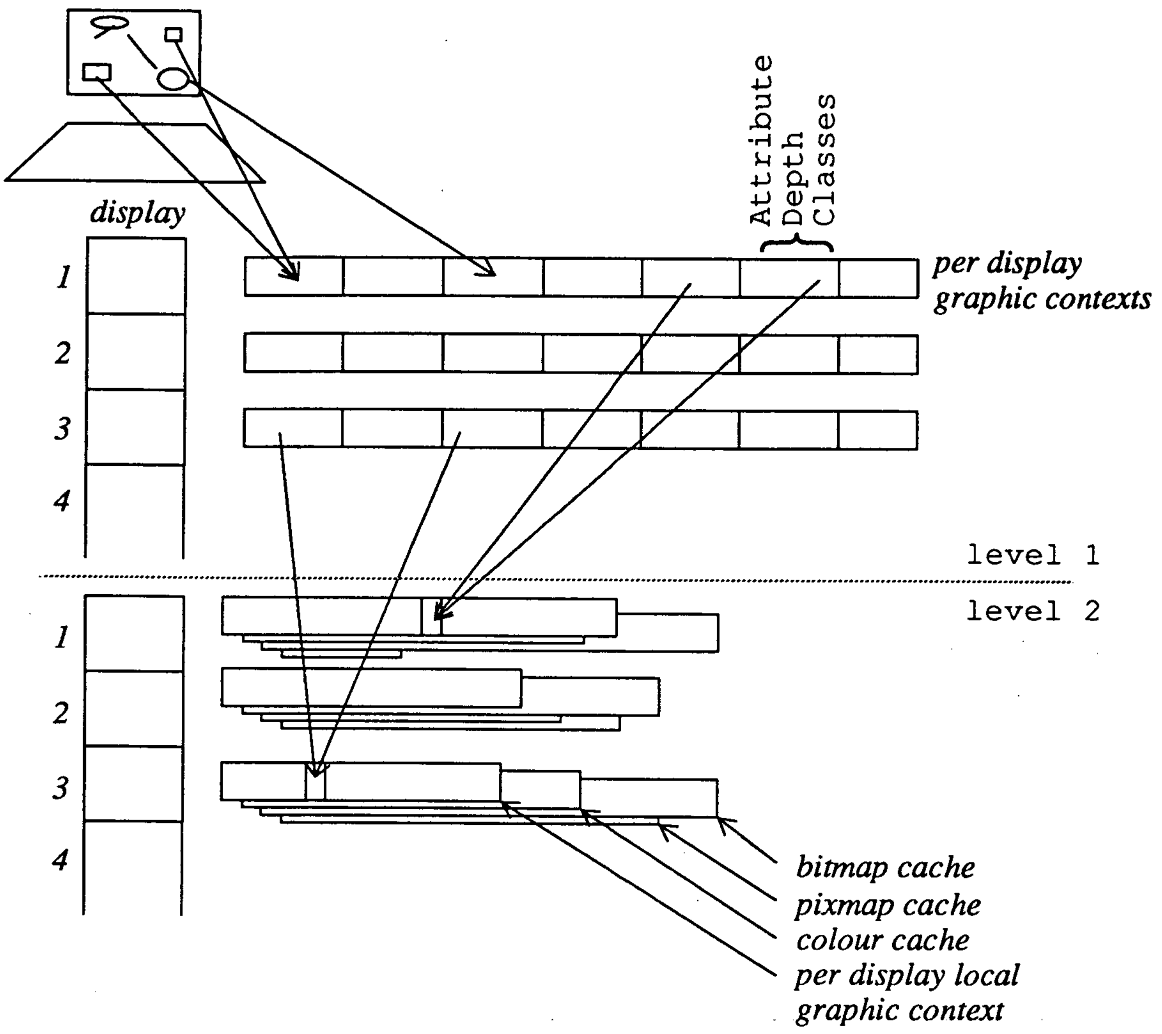


Figure 5-13. Two Level Hashing Attribute Management

Figure 5-13 shows, through a two level hashing structure, how both of these two problems are resolved. When an attribute needs to be used, it is 'locked', and when its use has been completed, (usually at the end of an application run), it is simply 'unlocked'. The algorithm is one of seeing if there is a local graphic context available and using it. If it is not present, a search is made to see if there is an identical local graphic context. If one is not found, then a new local graphic context has to be built. During this building process, existing identical colour, bitmap and pixmaps are used

to make the final local graphic context. Once a local graphic context is built, corresponding X server resources are also requested.

The discussion of text alignment is probably more appropriate in a separate section, but since it is a visual attribute, it is briefly discussed here. With any system that draws text automatically, there is difficulty in correct placement. The basic options provided by X, of simply drawing text horizontally left to right were considered not very flexible. Rather than a solution on an ad hoc basis, a number of requirements were formulated, the application of which are illustrated in figure 5-14. Firstly, text could be rotated in 45 degree steps, secondly left, middle and right justification could happen on both edges of a text strings bounding box, and finally, regardless of the transformations text objects pass through, the result must be readable in normal top-down, left-right flow.

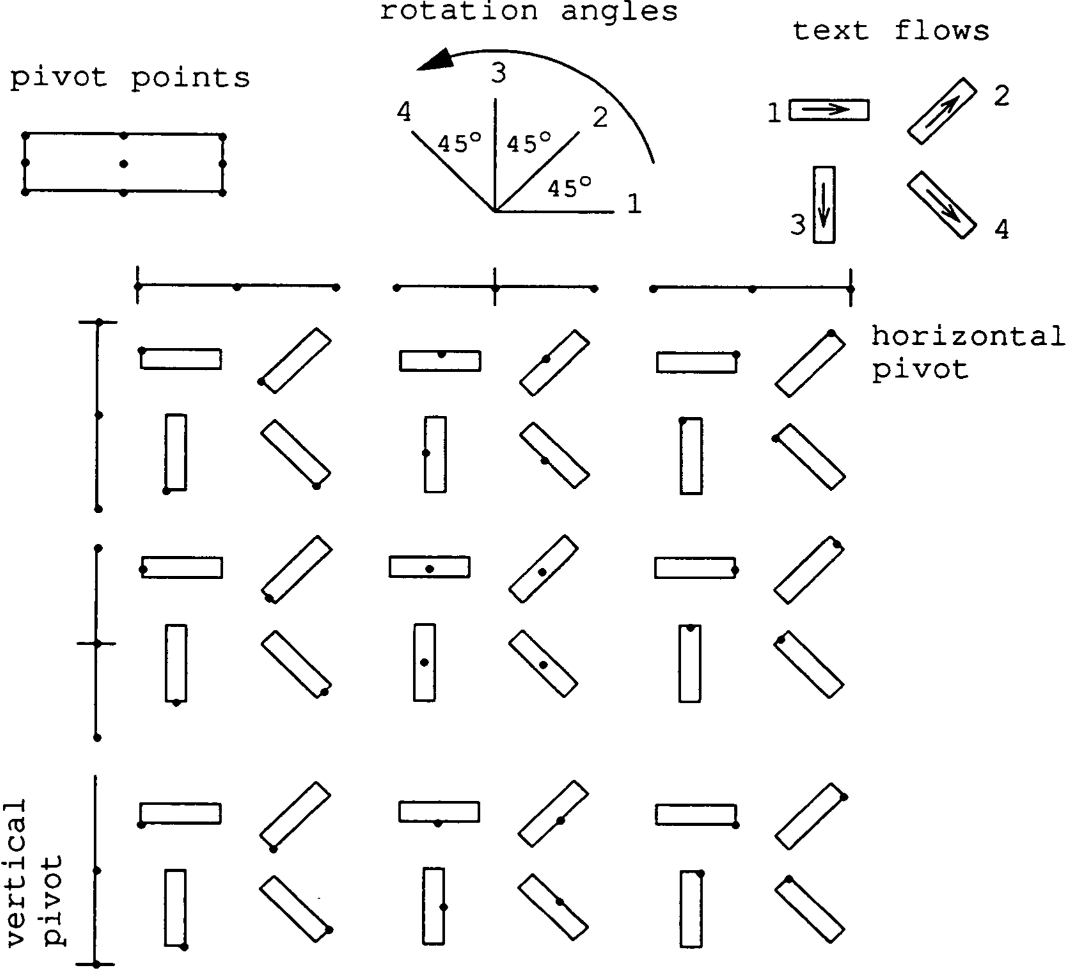


Figure 5-14. 36 Ways to Draw Text

Since the current text draw facilities are based on using bitmaps, the effect of rotate is illustrated in figure 5-15. This example reflects the bottom left batch of four in figure 5-14, illustrating the pivot point as the bottom left hand corner of a normal left to right text flow. The rectangular shapes reflect the shape of the character frames used in X.

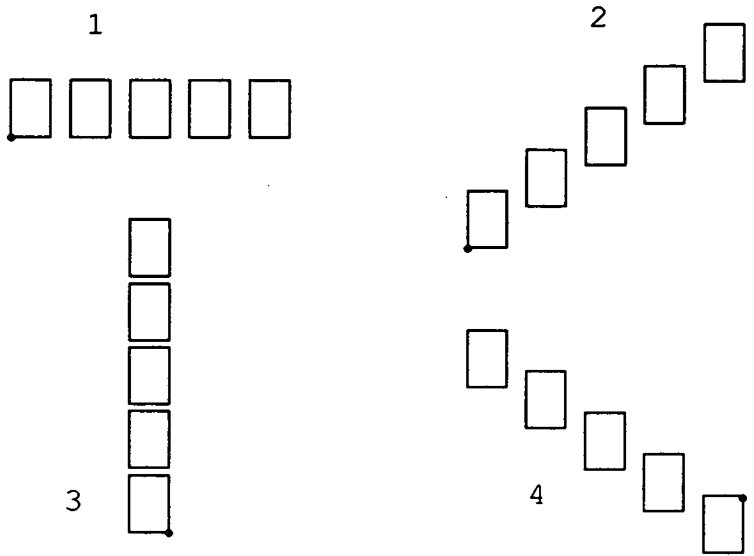


Figure 5-15. Rotated Bitmap Text

5.6 User Interface Management Services

At the time of development of SAGE, two user interface standards were beginning to emerge, namely OpenLook [51] and Motif. Each were designed to bridge the gap between application and user, by providing the programmer with concepts and tools to develop graphical user interfaces. For the development of SAGE, the choice has been immaterial since both provide standard user interface components like buttons, menus, scroll bars etc., to a reasonably equal level of functionality. In fact, (as also mentioned earlier in section 5.2), a mock-up version of SAGE was based on the XView Toolkit from SUN Microsystems.

There are a number of tools now emerging that help in the process of developing a user interface, Serpent [49] and Picasso [50] being two typical offerings. Serpent, in particular, provides a language independent layer in which the user interface is described. This scheme allows applications to be separated from the

display technology so that Serpent is able to provide C and ADA application interfaces. Unfortunately, this technology became available late in the development of SAGE. Consequently, the presence of the prototyping language UIL with Motif, gave it the edge over OpenLook. Nevertheless, the division between the Motif module and ADA module, is such that no Motif dependent features are present within the ADA module. As a result, it is possible to envisage being able completely to replace the Motif module with say an XView module, and require no changes to the ADA module.

This distinction has been achieved by following the same ideas now present in tools like Serpent, namely separating form from function. As mentioned earlier this chapter, the Motif and ADA modules communicate through a LISP like human readable syntax. As we have seen, by arranging for it to be human readable, the ADA module can be invoked as an independent application which allows for much easier testing and debugging. The fact that a parsing penalty must be paid, is mostly negated since the traffic between the two modules tends to be simple command/handshake sequences instigated by a user. By supporting a LISP style, arbitrarily complex data can be transferred. A good example of this, is the main menu bar popup, which is transferred to the Motif module at application start-up. Figure 5-16 illustrates part of the LISP syntax, with the resulting form of the main menu popups. Again, we note that the time consumed by this transaction is negligible, especially compared with the time consumed by the widget building activity that Motif in conjunction with the X Toolkit Intrinsics must undertake.

• *MOTIF-ADA dialog extract:*

```
...
    (
      cascade "Comms"
      (
        (
          cascade "Synthesise"
          (
            ("Simple" (comms synthesise simple))
            ("SimpleOpt"
              (UNIMPLEMENTED comms synthesise simple_opt))
            ("Opt" (UNIMPLEMENTED comms synthesise opt))
          )
        )
        ("Check" (comms check))
      )
    )
...
```

• *Resulting main menu bar definition:*

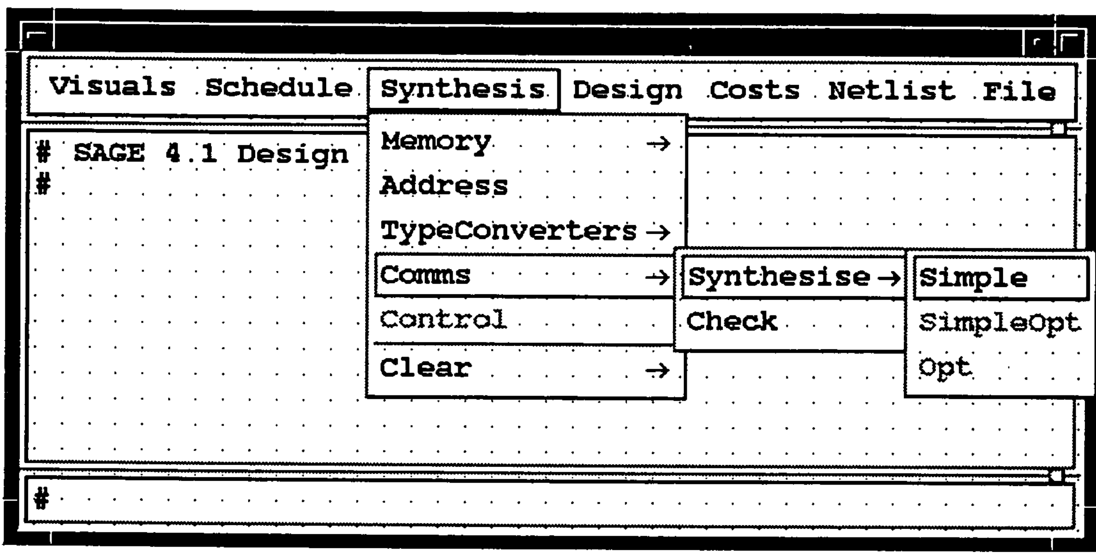


Figure 5-16. User Interface Component Construction Example

5.6.1 Programmers UI Interface

As odd a sub-section heading as it may sound when 'UI' is replaced with 'User Interface', the complexities of building software require general management functions which can act as an interface for programmers [26]. As with all management functions, they appear as red-tape, but are in fact necessary for the benefit they bring in the longer term. In the development of SAGE, there were upwards of fifteen active programmers, each developing their own solution to communicate textual information to a user. Rather than let this ad-hoc approach

become the normal behaviour, all the key requirements for a consistent and uniform means of communicating with a SAGE user were identified. A second reason for having this interface, was allowing its output destination to be controlled, without any changes being required in the modules that made use of the interface. For example, a user could turn all warning messages off, as well as the output sink could be made to recognise repetitive errors and only display a total count plus one copy of the repeating error message. Another example is that a message could have its output destination redirected to appear in a panel, depending on its importance.

Since the choice a programmer has is with a 'red-tape' interface on one hand, and a simple ADA'PUT_LINE' on the other, and no project requirement that such an interface is used, it is necessary to minimise completely the complexity of the interface to try and increase its appeal, while still achieving consistency and uniformity for a user. (It is sad that despite this interface being present before any significant programming effort had happened, no other programmer made use of this interface, even though all the correct design decisions, as described below, can be argued to have been made).

In total, six facilities are provided. At the heart of these facilities, is a general parameterisable message structure. This is composed of two parts, one being the static parts of the message, and the second being overlay arguments to that message. This is illustrated in figure 5-17. If one or the other is imagined to be null, then the full range from a fully static message to a fully dynamic message can be constructed. Having more lines and/or more parameters than actually present simply means that the remainder are appended to the end. If too few arguments are supplied, then the remaining even strings (counting from 1), are passed to the output. During this overlay process, the static and dynamic messages are forced to have the same number of lines by the implied addition of null lines. The facility to declare a message is called 'registering a message', while the facility to produce a message is called 'raising a message'.

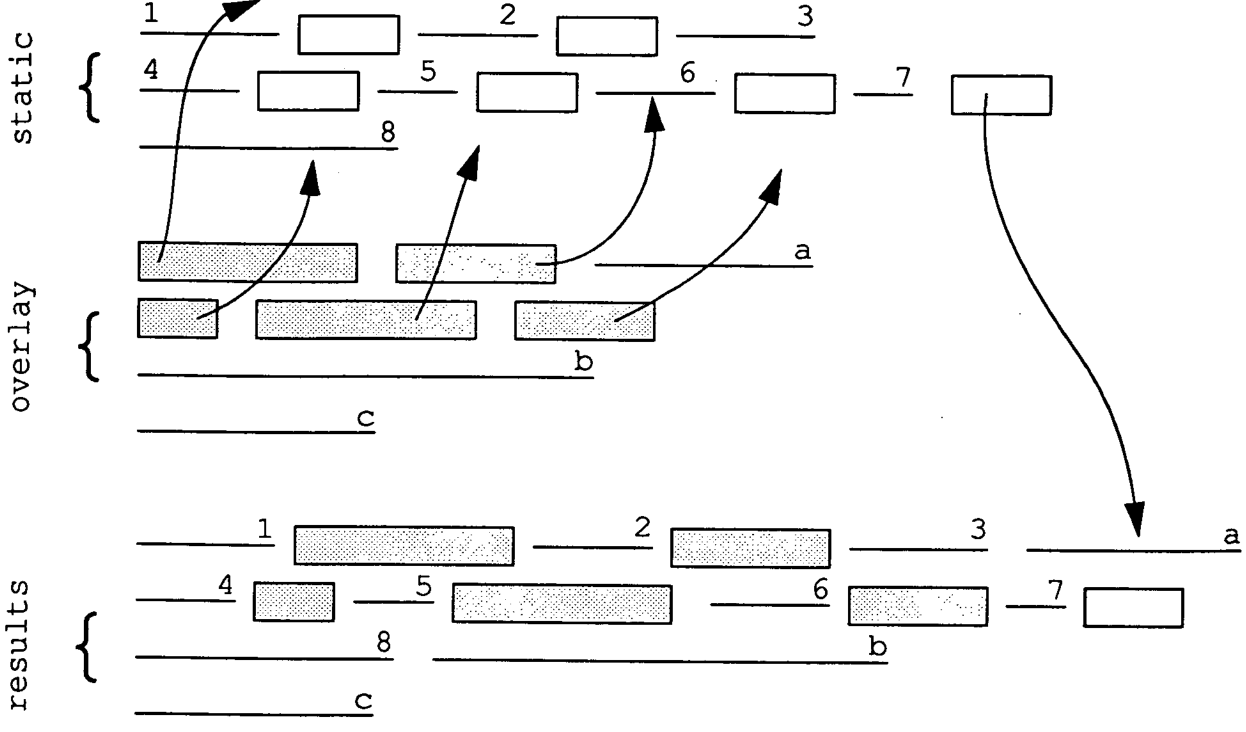


Figure 5-17. Stylised Message Construction Example

One immediate benefit of this approach is that of saving memory by defining static strings only once. There are a number or more subtle benefits. The static message acts as software documentation, since the arguments that would normally be overlaid can be actual meaningful strings, and this naturally leads to a dump facility. This can be compared with the way that a programmer doing the equivalent action in source code would have this information hidden, unless the nature of the arguments is obvious, or the programmer has had the patience and presence of mind to add a comment in the source code describing the nature of the arguments. Another benefit is that long 'list' style overlays can also be supported.

For all software systems, messages can usually be assigned a certain importance. Orthogonal to this categorisation is that of identifying messages for a programmers benefit as opposed to those for the users benefit. Rather than leave the programmer to add this information to their message, the requirements have been formalised. The

former can be specified from the enumeration:

`(UNDEFINED, NONE, INFORMATION, WARNING, ERROR, FATAL),`

while the later can be simply selected from the enumeration:

`(USER, INTERNAL).`

If internal messages are raised, a programmer usually requires to know if the problem is in their code, and if it is, where in their code it happened. Again, being such a common requirement, it can be formalised by having a facility to register software. Figure 5-18 illustrates such an example.

```
USER_INTERFACE_WHO_ID : WHO_ID_T;
...
USER_INTERFACE_WHO_ID :=
  REGISTER_SOFTWARE(
    SOFTWARE_NAME => "user_interface/errors",
    MAJOR_VERSION => 1,
    MINOR_VERSION => 1
  );
```

Figure 5-18. Registering Software

Although not programmatically enforced, the software name follows a hierarchical naming style. The version numbers are to help track code issues. The elegance of this approach used, is such that all three arguments could be furnished automatically by a tool like RCS or SCCS from UNIX. The handle returned is used in the process of registering messages.

When programmers raise a message in the FATAL class, the implication is that the algorithm can no longer continue. With this in mind, the facilities for registering and raising messages are in fact complemented by registering and raising exceptions. Raising an exception is identical to that of raising a message, except the thread of control is taken over by the catcher of the exception (usually the top level user interface software).

Message construction is illustrated to highlight how simple the job of creating a message actually is. As has have seen, messages are made up of lines, which are made up of words. Line and (multi) word constructor functions are available as: `l()` and `w()`. The later takes ADA strings, while the former takes the output of `w()`. Using the ADA concatenation operator, it is possible to form many line/word messages, e.g.:

```
l(w("bad element:")&w("<name of bad element>"))&
l(w("this element has no input or output arcs"))
```

The parameter arguments are by convention delimited by angle brackets. Raising a message, for this example which has only one parameter, simply requires this parameter to be supplied, e.g.:

```
l(w(OBJECT.BB.NAME.all))
```

The two remaining facilities provided, include a mechanism to get a simple okay/ cancel response from a user (called 'waiting'), as well as a way to report how long a particular action is estimated to take (called 'reporting'). The former is in fact not a desirable facility, since it introduces state information that really should belong only in the user interface code. Nevertheless, being a consistent and small abstraction, its provision is made in recognition that at times it can provide a quick solution to getting information from a user without having to wait for such requirements to become available directly through the user interface. Note, only raised messages can make use of this facility, by declaring the fact that it expects a reply. By having an explicit facility to report the estimated time of completion, is a simple way to display a bar on a users screen, and then change its size to reflect how much longer it is expected to take, (or, just as importantly, how much longer it is taking than was estimated).

Figure 5-19 shows some typical output. Note, because of the way these facilities work, it is quite straightforward to change the presentation style.

- Single line message

```
# (user/information) "schedule/check_graph" will take approximately 20 seconds
  (from user_interface/errors[1:1])
```

- Multiple line message

```
# (internal/error), (from schedule/check_graph[1:2]):
#    bad element: "BB245"
#    this element has no input or output arcs
```

- Message requiring user response

```
# (user/information) this is an irreversible action (from general/confirmer[2:1])
# input required for "OKAY_CANCEL" request, type TRUE or FALSE
true
```

- Message requiring user response, but given a 'bad' reply

```
# (user/information) this is an irreversible action (from general/confirmer[2:1])
# input required for "OKAY_CANCEL" request, type TRUE or FALSE
rubbish
# (user/warning) bad input, try again (from user_interface/errors[1:1])
false
```

- A dump of the *messages* and *exceptions* associated with each *software* item

```
*** general/confirmer[2:1]
  M message class : USER, importance : INFORMATION -
    | <null message>
*** schedule/check_graph[1:2]
  M message class : INTERNAL, importance : ERROR -
    | bad element: <name of bad element>
    | this element has no input or output arcs
*** user_interface/errors[1:1]
  M message class : USER, importance : INFORMATION -
    | <software> will take approximately <length of time> seconds
  M message class : USER, importance : WARNING -
    | bad input, try again
```

The dump process marks *messages* with `'M'` and *exceptions* with `'E'`, but this example has no *exceptions*.

Figure 5-19. Typical Output

As a minor technical point, the process of registering is an initialization activity, which means it should ideally happen during the *elaboration* stage of an ADA executable. In contrast, the processes of raising, waiting and reporting should occur during the main work phase of an invoked subprogram.

5.6.2 Rubberbanding

From window managers that can show the bounding box of a window being resized or moved, to object moving found in drawing packages, nearly every complex software tool has a requirement for rubberbanding. Rather than yet another ad-hoc approach seen in most of these tools, a much more general and systematic approach to rubberbanding can be defined. With such a framework, not only can simple cross hairs and rectangular region shadows be handled, but also more complex objects like parts of a schematic, where some ends of the shape being dragged can represent anchored net ends.

From an abstraction point of view, we have a window of `width x height`, a starting point relative to the origin of this window, `(x, y)`, and a starting shadow shape. As the cursor is moved, various deltas, `(dx, dy)`, may have a direct impact on the shape of the shadow. The shadow is made up of simple segments, defined as `(x, y)` pairs. The important observation, is that each end of a segment has a certain location at the start of the rubberbanding, and a certain behaviour with new delta values, once the pointer is being tracked. In fact, seven possible behaviours can apply in each axis direction, for the starting location. For a given `x` value say λ, the first three are destructively to replace this value with either the starting cursor `x` position, or the leftedge location of the window (i.e. `0`), or the rightedge location of the window (i.e. `width`). The next three are the same as these, but the given λ value is also added in. The seventh behaviour is to simply leave the given λ value as is. When the cursor is moved, a boolean associated with the updated `x` value determines if `dx` is added in or not. Obviously a similar scheme applies to a given `y` value.

As described, the scheme places no limits on the values involved - in fact, it is even possible for the starting cursor point to have negative values (from a `X Window System` point of view, this can only happen through use of something like `XSendEvent`). As a result, this is only half the picture. For every point in the shadow, there are four values, a pair for each ordinate, that can determine the limits on how far the given point can travel in a given ordinate. The seven behaviours for the shadow points, also determine the starting values for the limits. The limit values are

static and therefore do not track dx and dy changes, and are only paid attention if the corresponding boolean, saying that the limits should be tracked, is set. This is illustrated in Figure 5-20.

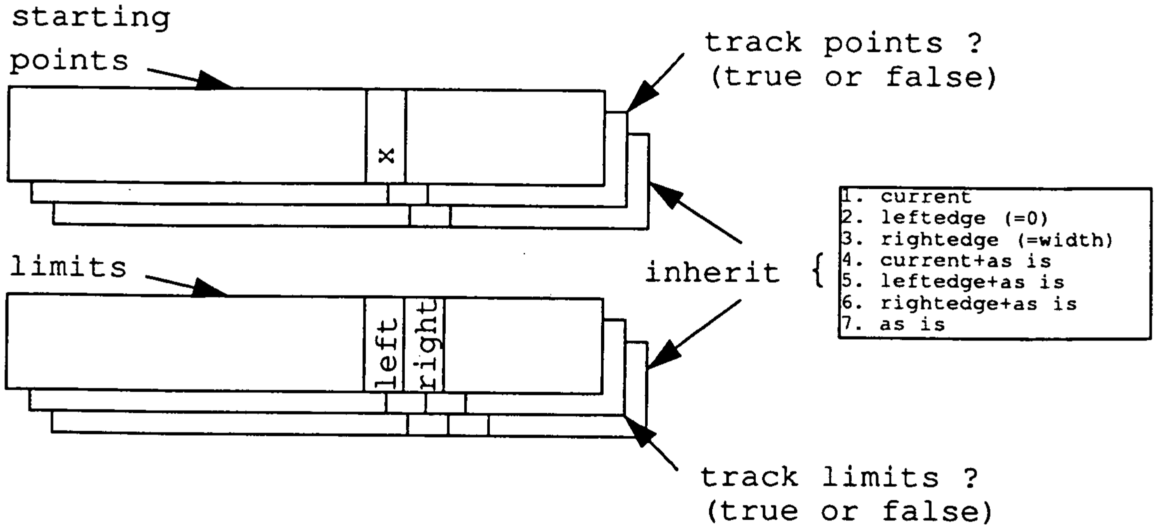


Figure 5-20. Tracking, Inheritance and Limits for the* x *Ordinate

From this basic behaviour, all the rubberbanding styles described earlier can be handled, plus much more. The classic problem with cross-hairs is that the object being selected can be partly hidden by the cross-hairs themselves. Now, by the simple replacement of the two cross-hair segments by four, and having a behaviour of current+as is for their endpoints, and selecting the associated starting point values as a five or so pixels, the problem is solved. Figure 5-21 (a) illustrates this, as well as showing how the properties relate to the segment endpoints. In this example it makes sense to place limits on the segments. A good example where no limits need be used, is that for a rubberbanded box shape defining a zoom area. Most graphical tools making use of this technique only use it for zooming in, but by supporting the rubberband image beyond the boundaries of the window, it can in fact also be used for zooming out, as is the case with SAGE. Figure 5-21 (b) illustrates the behaviours

associated with a rubberbanded box. For simplicity, both diagrams in the figure do not show the state of the limits - which would only apply to (a) in this example.

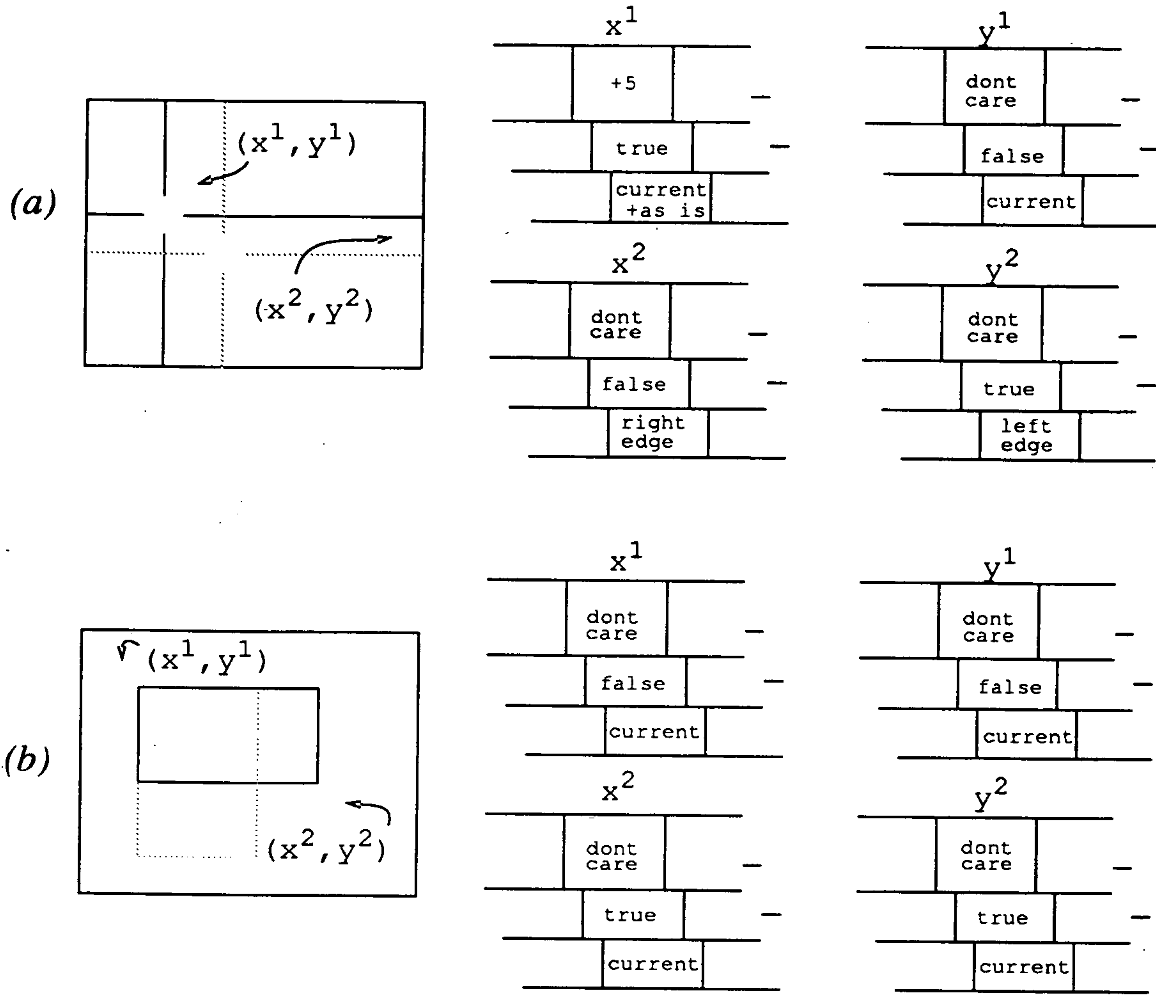


Figure 5-21. Rubberbanding Cross-Hairs and Box Examples - Points Only

For completeness, a number of minor practical details need to be enunciated. Since rubberbanding is an interaction issue, the `Motif` module is responsible for the job. Because of the possible complexity of a shadow, it is necessary to choose carefully the algorithm that tracks the cursor. With the `X Window System`, there are three broad ways to track a cursor, and the method chosen by SAGE is to make it the responsibility of the `Motif` module to query the location of the cursor. This has a round-trip communication cost, but ensures no shadow catch up effects that many other systems suffer from. The final comment is that the actual drawing and update of shadow happens when the server is grabbed, and by using the exclusive-or technique.

Your home electrical system is basically a bunch of wires that bring electricity into your home and take if back out before it has a chance to kill you. This is called a "circuit". The most common home electrical problem is when the circuit is broken by a "circuit breaker"; this causes the electricity to back up in one of the wires until it bursts out of an outlet in the form of sparks, which can damage your carpet. The best way to avoid broken circuits is to change your fuses regularly.

Another common problem is that the lights flicker. This sometimes means that your electrical system is inadequate, but more often it means that your home is possessed by demons, in which case you'll need to get a caulking gun and some caulking. If you're not sure whether your house is possessed, see "The Amityville Horror", a fine documentary film based on an actual book. Or call in a licensed electrician, who is trained to spot the signs of demonic possession, such as blood coming down the stairs, enormous cats on the dinette table, etc.

-- Dave Barry, "The Taming of the Screw"

6. Foundations

The development of any successful system, whether in the abstract, like software, or physical, like house building, requires the right tools, techniques and concepts to ensure both the construction phase and the operational phase meet the desired objectives. If the techniques and concepts form the substance of a design, then the tools can be said to form the foundations. Although many aspects contribute to the success of a design, having the correct foundations is one of the more important aspects, since poor foundations can easily limit what techniques and concepts can be consequently developed. There are many examples from the physical world where the wrong tool is used. From the absurd, like using garden shears to cut a persons hair, to a surgeon using a kitchen knife for delicate eye surgery, to more subtle examples like using the back end of a screwdriver to bang home a nail. The point being, that in real life the disparity between a tool and its application is generally obvious, but this is not the case with software. The net result is, the range of software solutions can range from practically useless, to nearly perfect, all as a result of the form of the foundations.

The consequence of applying tools, techniques and concepts, can in a general sense lead to new tools. In many ways this is the life cycle of SAGE. But, rather than a simple combination of ADA [3] as the tool plus ideas to produce SAGE, levels of abstraction can be identified which build higher level tools, which form the steps that lead to the application SAGE. This chapter concentrates primarily on the requirements

of the first level. This first level, being the next step after the language itself, is in fact the most general. It is also very important, in that mistakes, anomalies or limitations not only impact the remainder of the levels, but can also require a monumental effort to correct. The overall quality of the first few levels has a gearing effect. If it is slightly bad, then things could become very confusing at higher levels, while if its good, it will help accelerate the development and/or the performance of higher levels.

The `X Window System` [57] is a good illustration of such successive steps from protocol to application interface as shown in figure 5-5 on page 129. This figure also illustrates one of many flaws in the `X Window System`, namely what could be termed 'abstraction leakage'. Rather than each step being a complete abstraction, there is a requirement on a user to understand all the proceeding steps. This is rather like a car driver being required to understand the workings of an internal combustion engine, since occasionally it becomes the drivers responsibility to sequence the firing of the spark plugs.

As Stroustroup [9] has indicated, over many years, the emphasis in developing software has shifted from the design of algorithms to that of how to partition a problem. This directly addresses the limitations of the intellectual capability of humans as discussed by `Booch` [7]. `Booch`, in recent years, has probably been the most prominent proponent of identifying general abstractions and then developing and formalising them. Interesting as his treatment is, there are a many flaws in his handling of the subject matter as described in *"Software Components with* ADA*: Structures, Tools, and Subsystems"*. The most significant flaw is that while acknowledging the emphasis is on developing well engineered interfaces, he contradicts his own advice, in developing interfaces that are not proper encapsulations of an abstraction. Rather, they are like the `X Window System`, suffering from abstraction leakage. With his taxonomy of objects, he identifies an explosion of 501 separate interfaces, which forms an immediate barrier to a programmer trying to choose a suitable interface. Being a recognised authority on `OOD`, or object orientated design, it is interesting how some encapsulations fail in this regards. A good simple example is adding an element to a `Booch` list. If we have two variables, `a` and `b`, that refer to the same list, then adding an element to either list should mean that both still

represent the same object. Instead, if an item is added to the head of a, then a and b will no longer refer to the same object. These, plus several other problems with the Booch interfaces, have caused significant problems in the development of SAGE. Another flaw with his approach, is that his interfaces do not address the equally important issue of ensuring that the algorithms behind interfaces are also optimal.

This chapter does cover similar ground to Booch, but with much closer attention to the development of a well engineered interfaces supported by an implementation that is also near optimal. Unlike the taxonomy of components identified by Booch, the driving influence in the development of components described in this chapter, has of course been the requirements of SAGE, but with a view to the development of general foundations with much wider applicability. Only where an improvement or new approach to the design of an algorithm over published information is made, is it discussed.

In summary, the components have been designed to represent good examples of flexibility, useability, simplicity, elegance and performance.

6.1 Lists

The biggest problem with many current implementations of the list object, is that they are historically engulfed in a quagmire caused by the concept of lists used in LISP [10]. This really has been the root of all problems to many so called abstractions, since they immediately require a programmer to understand the internal structure of a list to be able to do useful things with it. The main features of LISP that pervade the abstractions are the head (CAR), and tail (CDR) operations. A closer analysis reveals in the days when memory was measured in bytes (as opposed to Kbytes, MBytes etc.), the singly linked list implementation apparently was the most space saving. This also meant, the very common requirement of adding items to the end of a list, became an O(n) operation. Consequently, the language constructs had an implied way of saying that this was inefficient - for example, the notation to get to the fifth element of a list would look like this ridiculous construction: CDR(CDR(CDR(CDR(CAR(list))))).

The surprising fact is, with a simple modification, the same memory requirements can lead to `O(1)` behaviour to add items to the head *and* end of a given list. The mechanism is simply to convert the list into a circular list. There is nothing new in having circular lists, but this time, the reference into the list is the last element in the list. Compare this with the normal approach of having to create an additional reference to the last element. Not only is adding to the end of a list now `O(1)`, but the same symmetrical algorithm can be used to combine two arbitrary lists in `O(1)` time. This algorithm is illustrated graphically in figure 6-1. (This same algorithm is used in the generic list package for C. A doubly-linked version is used for the generic ADA package. Both are discussed later in this section.) Also in the first part of this figure, are the three styles of list construction discussed, of which the third is used.

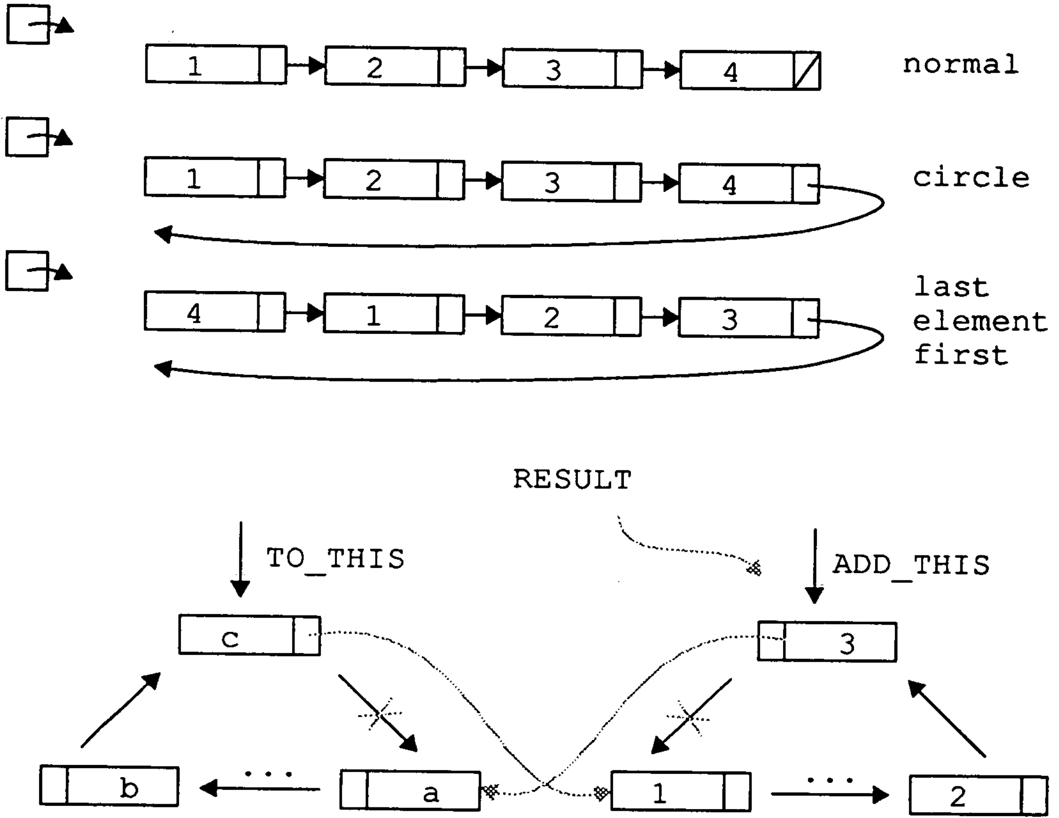


Figure 6-1. List Forms and O(1) Symmetric List Combine

Although performance is important, what is more important is the abstraction describing list behaviour. Thus, only if an implementation is radically inefficient should it direct the form of the abstraction. The direct implication of these statements

is that an implementation can be upgraded to provide better performance at a future date with next to no impact on the abstraction that a user sees. It is this emphasis on what makes life easier for a user, that has directed the development of the SAGE lists (and other) generic packages.

List objects can be 'created' and 'destroyed'. (These activities are distinct from creating and destroying the elements of a list.) In the implementation these two activities might be 'null' actions, but for orthogonality with more complex objects that can be created and destroyed, they are necessary. In SAGE, a list object is implemented as a reference to a record that contains information about the list and its elements. This gives the list the important property of multiple view consistency. So, several variables that reference the same object, all see the changes if the object is modified in any way. This is illustrated pictorially in figure 6-2. Other implementation details are that accesses to successive elements, backwards or forwards in a list, are O(1), and, since the last location visited is preserved, on average, random accesses to the elements of the list will approach being nearly 4 times faster than a simple singly linked list.

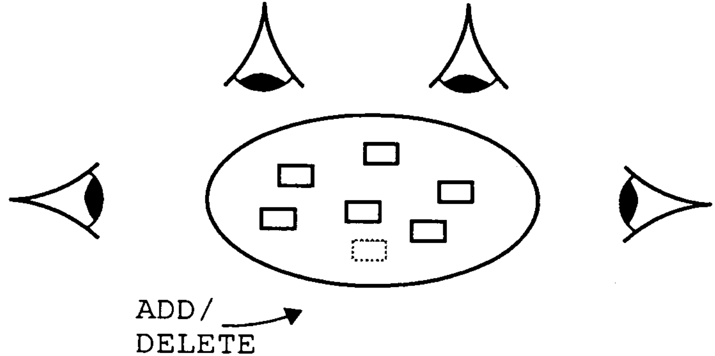


Figure 6-2. Multi View Stability

Again, from a user's point of view, visiting the nth element of a list, for viewing, addition or deletion appears reasonable. It is instructive, therefore to compare how the implementation of SAGEs iteration over a list, compares with that of Booch.

This is shown in figure 6-3. There are three points that can be made. Firstly, the Booch iteration exhibits poor locality of code. Not only can the 'TAIL_OF' assignment be missed, but it can easily be followed, inadvertently by more code.

Secondly, the Booch approach is not easily amenable to algorithms that need to reference list items like i-1 and i+1. The third point, is the most damning, in that making the reverse traversal from the final element to the first, is non-trivial both to express in ADA, and efficiency. With the SAGE lists, all it requires is the addition of the little word 'reverse' after the keyword 'in'. This last observation, combined with the fact it is easier to construct lists by adding items to the head of a Booch list, is the reason why so much software seems to produce output in the wrong (i.e. reverse) order.

```
for i in 1..SIZE(list)
  loop
    <some function of GET(list, i)>                     } SAGE
  end loop;


while not IS_NULL(list)
 loop
   <some function of HEAD_OF(list)>                     } Booch
   list := TAIL_OF(list);
 end loop;
```

Figure 6-3. SAGE ***vs*** Booch ***iteration***

Supplementing these basic list operations, are numerous templates. These take as arguments functions or procedures that either return a logical value or do something as a result of the data element being examined. A good example is that of the DESTROY_TEMPLATE, which recognises the fact that data stored in a list can be complex, and therefore a procedure to destroy a list should also be able to destroy the data elements that form that list. Another example is IS_MEMBER_TEMPLATE, which traverses the list to see if a given data item is present in the list. Figure A-1 on page 207 is the specification of the ADA list generic, and illustrates the other subprograms that complete the abstraction.

One of the reasons why such abstractions are so much easier in ADA than in C, is because of the 'generic' construct. Nevertheless, it is possible to emulate similar behaviour in C, by use of the macro preprocessor. The main complication, is the need to identify the form needed for the list specification that can exist in a user.h file, and the body part that will exist in the user.c file. Figure 6-4 illustrates how this is achieved in practice, by showing only a simplified ListAdd operation. Note, the

algorithm shown in the `list.h` file, is the same operation shown graphically earlier, in figure 6-1 on page 158.

list.h:

```
#ifndef _LIST
#define _LIST

/*---------------------------------------------------*/
#define ListAddSpec(ListAdd)\
extern void ListAdd(/* ToThis, This */);\          } spec

/*---------------------------------------------------*/
#define ListAddBody(ListAdd, Data)\                } body
void ListAdd(ToThis, This)\
  Data **ToThis; /* pointer to the list */\
  Data This;\
{\
  /* (1) allocate the new list element and init */\
  Data *NewListPtr = (Data *)malloc(sizeof(Data));\
\
  *NewListPtr = This;\
  NewListPtr->next = NewListPtr; /* set it up as a loop */\
\
  /* (2) add it to the end of the list */\
  if (*ToThis == (Data *)0) { /* simply add in the item */\
    *ToThis = NewListPtr;\
  } else { /* we add it to the end of the list */\
    Data *ToThis_ = *ToThis;\
    Data *This_ = NewListPtr;\
\
    Data *ToThisNext_ = ToThis_->next;\
    Data *ThisNext_ = This_->next;\
\
    ToThis_->next = ThisNext_;\
    This_->next = ToThisNext_;\
\
    *ToThis = This_;\
  }\
}
#endif /* _LIST */
```

user.h:

```
#include "list.h"
...
struct speedH {
  struct speedH *next;
  Ella_vtList vtlist;
  struct speedH *speedHlist;
};
typedef struct speedH SpeedH;
typedef struct speedH *SpeedHList;
ListAddSpec(SpeedHListAdd)
...
```

user.c:

```
#include "list.h"
...
ListAddBody(SpeedHListAdd, SpeedH)
...

ella_waveform_process() {
  ...
  if (currentVspeed == (SpeedVList)NULL) {
    SpeedV speedV;

    speedV.count = 1;
    speedV.speedHlist = (SpeedHList)NULL;

    SpeedHListAdd(&cn->speedHlist, speedH);
    break; /* out of the while loop */
  }
  ...
}
```

Figure 6-4. Generics in C

6.2 Trees

As Sedgewick has remarked [11], the property of a searching algorithm to exhibit guaranteed logarithmic behaviour for all searches and insertions is present in red-black trees, and its use can be justified whenever bad worst-case performance simply cannot be tolerated. The underlying implementation of the SAGE red-black trees is very similar to that as given in [11], but with the addition of a logarithmic performance delete operation as well.

As with the lists object, the basic red-black tree can have objects created and destroyed. For a created object, there is again viewing, addition or deletion facilities. This time, the requirement to traverse the list can be more elaborate, since the start

and finish points can be keys. In fact, ten different ways to do an in-order traversal of the tree are provided. This, and the other subprograms used are shown in figure A-2 on page 211.

Red black trees are fundamentally a form of binary tree. Binary trees also have useful properties, in particular their ability to support all the three main graph traversals, namely pre-order, in-order and post-order. This is illustrated in the binary tree ADA specification shown in figure A-3 on page 213. In particular, note the way the traversal can be started with an arbitrary key value.

This same approach can be extended to general tree structures. In this case though, the requirements for traversal are restated as *breadth* and *depth* walks instead. This is shown in figure A-4 on page 215. The binary and n-ary tree routines form the heart of the naming software described in section 3.2.3.1 on page 49, to support arbitrary length strings with O(log n) string match behaviour.

In a more historical perspective, the red-black tree generic was developed after those for the binary and nary tree, and this is reflected in the richer set of operations provided, particularly with the presence of iterators.

6.3 Rectangle Management

The ability to manage efficiently rectangular objects is essential for many of the graph applications discussed in this thesis. The primary aim is to provide efficient stored state devices that have good behaviour for point and region queries, as well as efficient manipulation facilities such as add and delete. For spatial data there are two classes of algorithms that can be used, namely quad trees and 4-d trees [53]. As shown in [58], a variation on quad trees that no longer maintains bisector elements in a list, but instead assigns it to a quadrant depending on the coordinates of one of its corners can satisfy all these requirements provided a limit is placed on the number of objects stored in a quadrant. Thus the only pathological case that can cause the data-structure to depart from O(log n) behaviour is when there are many rectangles that are coincident, since they must all be stored in a list in the same quadrant.

It should be appreciated that the use of rectangles is really an approximation for different sorts of objects that might be presented to a user in a visual. In particular, encapsulating objects like lines by their enclosing rectangles can imply a region far greater in size than actually crossed by the lines, particularly if the lines are not orthogonal to any axis. Thus the list of rectangles returned by a point query would have to be rescanned using a slightly higher fidelity algorithm as defined by the actual object represented by the rectangle. Thus, in the case of lines, this is either the distance of the perpendicular to the line or to the nearest line end point if the given line has to be extended to form the perpendicular.

The whole approach of quad trees is based around the successive dividing of an area into four quadrants as soon as the number of objects stored in the given area exceeds a certain threshold. As mentioned earlier, the problem is how rectangles that do not fit into the quad tree hierarchy are managed - i.e. the bisector elements. Figure 6-5, illustrates how this is achieved. The first two parts, (a) and (b), represent the simple approach of simply storing the bisector elements at the largest enclosing quadrant. The next two parts, (c) and (d), show where the bisector elements are placed based on their bottom-left hand coordinates. The important point to note is that the area represented by the rectangular bounding box of all the objects in each quadrant, can be much greater than the area actually delineated by the given quadrant.

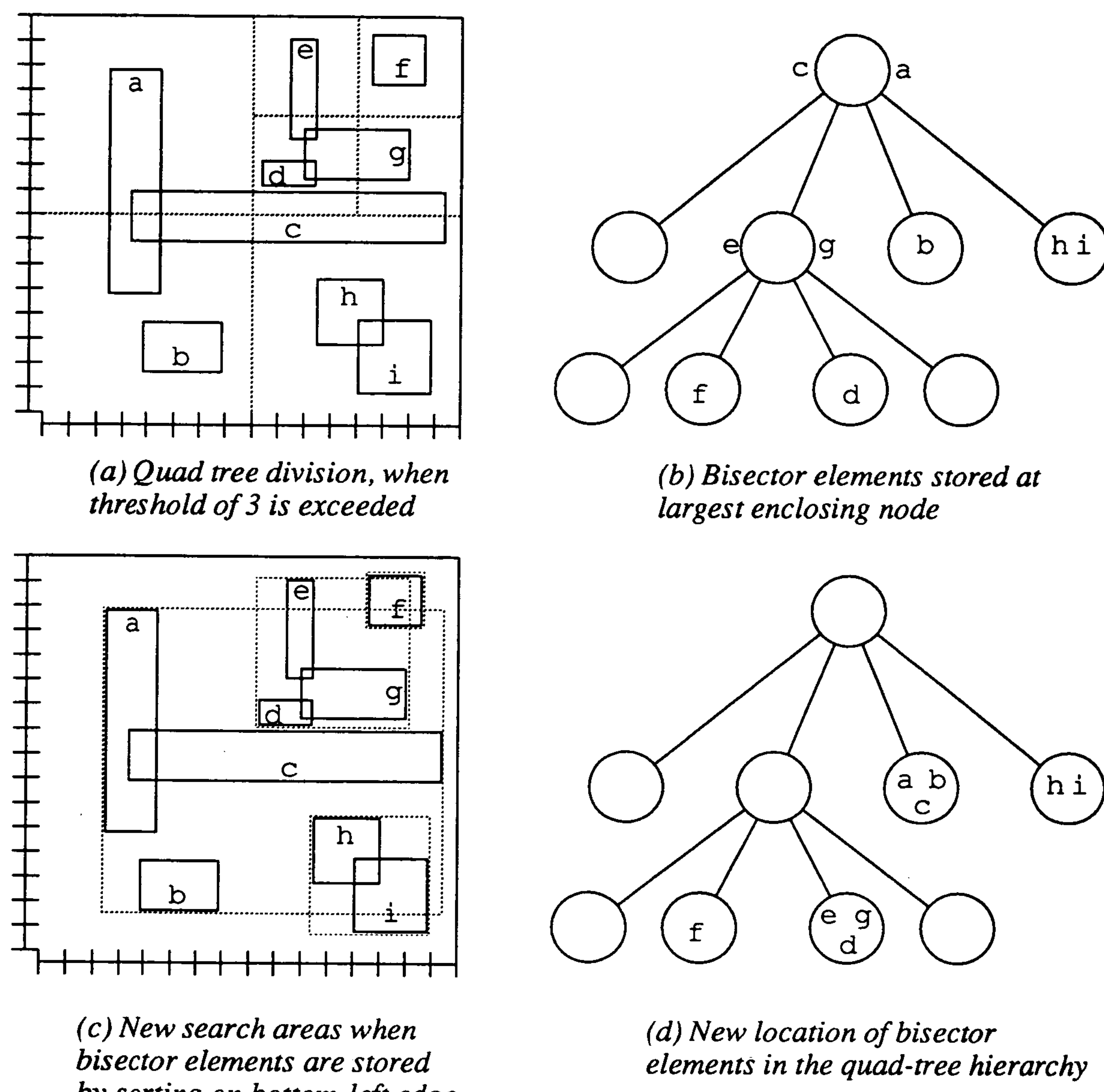


Figure 6-5. Quad Tree Storage - Bisector Element Management

Whereas the main advance on [53] is the specification of a generic that can provide rectangle management in its most general form, there are a number of equally important improvements and extensions to the original algorithms. The generic has been designed to handle regions not only defined by integer space, but also floating point numbers. The maximum quadrant size is therefore implied by the numeric types, while the smallest quadrant size can also be specified as an argument to the generic. As with the other generics, a data element is added or deleted from a

rectangles database, and therefore the function that determines the extent of the object can be passed as a parameter to the generic. As well as the basic object addition, deletion and housekeeping facilities, the generic provides iteration facilities that form the basis of the region/point query facilities. Figure A-5 on page 219 contains the actual rectangle manager ADA specification.

The main advances in the algorithms relate to reverse or backward quadrant creation, region bounds update during delete operations and detail culling. If the first quadrant is very big relative to the picture elements being drawn, then as items are added, a well trodden daisy chain of quadrants will be created, and always traversed. To remove this inefficiency, a 'best guess' can be made as to the size of the picture elements being drawn, but this leads to a greater chance of a picture element being bigger than the top-level quadrant. Consequently, reverse quadrant creation happens. Even this is not sufficient since numeric edge effects can lead to reverse quadrant creation failing. As a result, at this stage, such objects are placed automatically in the 'too big' class and treated specially during rectangle database queries.

During addition of objects into the rectangle database, the maintenance of the bounds at each quad tree node is straight forward. When deletion happens, the process is more complex, since it is not sufficient simply to subtract the area being removed since other regions may overlap the given area. Note, this problem does not arise if bounds had to be within the actual quadrant size. The algorithm to update these bounds during deletion is to backtrack at each quad tree node with a newly computed sub-quadrant bound and recompute that node's bounds by using this new value and the bounds associated with the remaining three sub-quadrants. Note, this process happens in step with the normal requirement to replace quadrants whose sub-quadrants are basically terminal, by terminator quadrants themselves. This process of replacement is itself complicated by ADA, since it prevents the declaration of unconstrained variant types, requiring that the parent of the quad tree being replaced having to be visited in order to delete the non-terminal quadrant and then create the replacement terminal quadrant.

With the vast amount of information that can be stored in a rectangles database, it is apparent that a general draw action can lead to the typical problem exhibited by many drawing systems of spending minutes if not hours to update a display. The heart of this problem is time spent drawing detail that is beyond the resolution of the display surface - whether it is a monitor (72dpi) or laser printer (300dpi). The technique of defining a cull region is specified in the rectangle manager iterator generic called 'ITERATE_LIMIT'. Rather than simply discarding what is beyond the cull limit, a parameter function called 'INVISIBLE' is called for each region which contains detail beyond the cull limit. As an argument to this function, is the actual coordinates of the region that contains the hidden detail. This means that an application can indicate on the display surface, by use of another colour for example, those parts that are too detailed to display. An application that makes use of this feature is illustrated in figure 6-6.

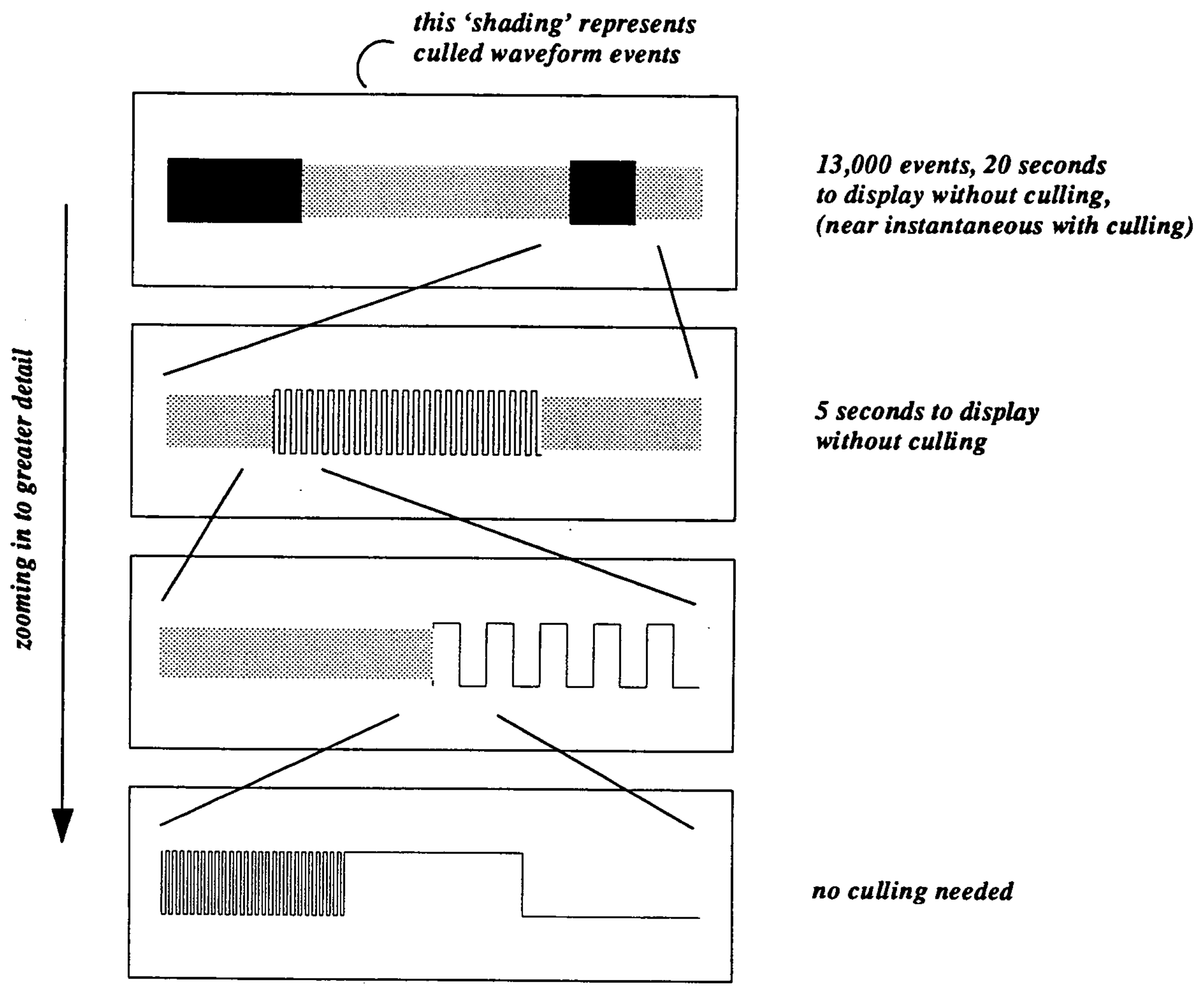


Figure 6-6. Waveform Culling Using The Rectangle Manager

This example represents a solution to one of the most serious flaws with existing waveform display systems. In fact, it also solves a second, related problem with many waveform display systems. Firstly, when zoomed out on a complex waveform, a designer can instantly see where there is hidden detail because a different colour is used in the regions returned by the function `INVISIBLE`. Secondly, for a typical workstation display surface, regardless of the complexity of the waveforms, the update will always be in seconds rather than hours. Despite this significant improvement over existing approaches, the example really requires only one dimension of the rectangle manager, and does not make use of the fact that during the database creation stage, the events will always be ordered. Another common feature of waveforms is the very common presence of repeated patterns - especially as shown

by (possibly gated) clocks. By noting these additional features, algorithms have been developed that provide orders of magnitude of improvement in waveform database creation and display, compared to commercial systems. Application of these ideas, as shown in [27], has in comparison with ELLAVIEW [34], produced an estimated x1600 increase in performance for database creation. Since waveform display facilities were not directly required by SAGE, the techniques adopted to achieve this performance gain are not described in this thesis.

SAGE only made use of the rectangle manager generic for storing visual information. As can be seen, its actual application area is far greater. Two additional examples will be given to illustrate this point. Firstly, the database can be used for routing algorithms, by providing very optimised route collision detection - a useful way to extend the schematic generation software. Secondly, complex map information can be captured in a single database. Through the use of the culling operation, it is possible to construct an approximation to the path delineated by the maps in the database, regardless of the magnification factor in operation.

It is noted in passing, that if there was a requirement within SAGE to store complex bitmapped images - that is images whose detail is generally well beyond that displayable on a normal workstation display surface, then a similar quad tree data structure based on average pixel values within each quad tree node could be used to provide efficient culling and a close approximation to the image to be displayed. The extra memory consumed by the quad tree data structure would, on average, be well offset by the gain in space efficiency resulting from storing relatively large unchanging screen areas in terminal quad tree nodes.

6.4 Sparse Sets

The simple action of a user selecting all the objects in an area can return a large number of selections. With many visuals, the ability to support multiple selection sets associated with each active visual can give a designer greater control but leads directly to an information management problem. With these sets of information, simple filtering actions and operations can easily lead to a rich set of activities that can

empower a user. For example, a group selection in a resource-time graph can be filtered to leave only function calls. Another example would be taking multiple selections sets from various schematics and combining them to produce a single set representing the components selected, ready for an action like displaying information about each component.

At the heart of all these requirements is a need to associate a unique number with every data model object and then apply set operations to achieve the user requirements. (Incidentally, the SAGE data model did have such a unique number associated with every data model object, but since every object was generally viewed through a reference object, the reference itself could have been used as the unique number. This would work since all the reference numbers are held in a single address space, and can be cast using unchecked conversion to an equivalent number representation - usually 4 bytes to 32 bits signed). The generic that achieves these simple set activities is shown in figure A-6 on page 221. As can be seen, not only are the basic set operations such as union, intersection and inverse set operations provided, but also operations that can add and remove elements or ranges from a set. To complete the operations, some set iterators are also provided.

The first point to note is that the set package understands about the '*world*' set, which can be defined to be some subset of the entire world as defined by the number range SET_VALUE_T'RANGE. Secondly, the package can handle enumeration types as easily as actual integer types. Thirdly, and most importantly, the generic has been designed to be fast and efficient. This last point is in sharp contrast to Booch sets, where the simple requirement of inverting a set is *not* provided, simply because it would be computationally very expensive.

This generic achieves its performance by storing contiguous elements in a set as pairs of values, namely the first and last in each contiguous set. This significantly complicates the process of combining sets - in particular, the union and intersection operations. These two operations, combined with the inverse operation, form the basis for all the other set operations. An additional complication is having to handle the edge conditions carefully. This implies that a set composed of pairs of values that can

be combined (e.g. `([b..c],[d..g])` transformed to `([a..g])`), cannot directly rely on the use of the successor and predecessor functions available in ADA since actions like `SET_VALUE_T'SUCC(SET_VALUE_T'LAST)` which might happen, are erroneous.

To illustrate the algorithm developed, the union operation will be looked at in detail. The intersection algorithm is very similar in style, and therefore will not be analysed. As implied earlier, the set object is stored as an ordered list, with each element in the ordered list being a pair of numbers defining a range. Whereas it would be an easy option to allow overlapping ranges (e.g. `([g..k], [i..z])`, as well as consecutive ranges that could be merged but are left unmerged (e.g. `([a..d], [d..e])`, the algorithm ensures neither case happens. Since the algorithm ensures that this cannot happen, its correct operation also relies on these conditions. Given two sets, the algorithm intertwines the two sets, by combining one with the other, and then the other way around when certain conditions have been met. The algorithm works by continually repeating two stages, firstly deciding which of the two lists is the pivot list, and then changing the pivot as needed to produce the largest contiguous range to add to the union list. Each of these two stages corresponds to a loop. Figure A-7 on page 224 contains the fully commented algorithm. Note, the list element is termed a slice. Also note, the three cases which need to be considered in order to decide if the pivot list needs to be swapped over in the second stage of the algorithm, as indicated as comments in the algorithm. The slice called '`THE_MERGE`' contains the element that is continually extended by the inner loop, until the two lists no longer can be merged.

Although designed with selection management in mind, again, as with the earlier described generics, the set abstraction is very general. There are many application areas where objects need to be treated as sets, and this abstraction could improve the performance considerably. Note though, the worst case performance of these algorithms tends towards that usually found in list based set implementations, since each slice will become a container for only one value.

6.5 Conversion Manager

As discussed in section 3.2.2 on page 44 and section 5.4 on page 133 concerning attributing for library creation and the rendering model respectively, there are many occasions where an object needs to have associated with itself, some third party information. This happens through the use of hooks. Where as the requirement has already been discussed in section 3.2.2, the implementation actually has a few subtleties that are worth discussing. At the heart of handling hooks is the need for unchecked conversion. This is a dangerous operation and therefore any use needs special containment. This does not remove the danger, but, as with any strongly typed language aims to do, it limits the potential scope for errors.

There are in fact two general ways that this indirection can be encapsulated. The first does not in fact need the use of unchecked conversion. Instead it relies on the data structures to which general hooks are to be associated to be converted to a generic that takes as arguments the hook data structures. There are a couple of flaws that make this apparently correct approach not acceptable. The most significant deficiency is that changes in the data structures associated with the hooks also require a recompilation of the generic taking the hook arguments. The other problem is that only a single instantiation of the generic can be easily supported, making multiple outlets such as needed when netlist generation (section 3.2) and schematic generation (section 4.8.4) are implemented within a single application much more difficult to achieve.

The abstraction used, although more complex, overcomes these problems. Figure 6-7 graphically shows the structure and interrelations between the data structures, the hook converters, and the user fields that form the hooks.

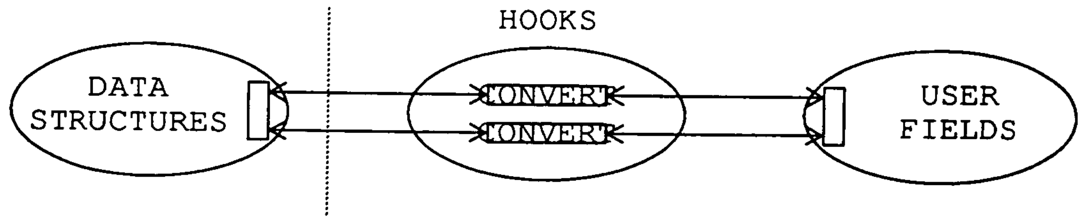


Figure 6-7. Complex Conversion Management of Data References

Both 'HOOKS' and 'CONVERT' are generics. By having these two levels, a more rigorous encapsulation is achieved. The contents of the former has already been discussed fully in netlist generation, with figure 3-11 on page 46 showing what the hooks generic looks like. The specification of the second generic is shown in figure 6-8, and demonstrates the three core functions that are required to map between the data structures and the user fields: PUT, GET and a destroy template. Through the careful definition of the arguments, only reference types can be converted. This ensures no inadvertent mapping between inconsistent types, which some ADA languages will support even though the conversion is apparently non-sensical, (e.g. converting a reference object to a record object).

```
with  UNCHECKED_CONVERSION;
generic

  type USER_REC_T is limited private;
  type USER_PTR_T is access USER_REC_T;
  type HIDDEN_USER_REC_T is limited private;
  type HIDDEN_USER_PTR_T is access HIDDEN_USER_REC_T;

--g-----------------------------------------------------------------------------g--
package CONVERT is
--g-----------------------------------------------------------------------------g--

  function CONVERT is new UNCHECKED_CONVERSION(USER_PTR_T, HIDDEN_USER_PTR_T);
  function CONVERT is new UNCHECKED_CONVERSION(HIDDEN_USER_PTR_T, USER_PTR_T);

  procedure PUT(THIS : USER_PTR_T; IN_THIS : in out HIDDEN_USER_PTR_T);
  function GET(FROM_THIS : HIDDEN_USER_PTR_T) return USER_PTR_T;

  generic
    with procedure DESTROY(THIS : in out USER_PTR_T);
  procedure DESTROY_TEMPLATE(THIS : in out HIDDEN_USER_PTR_T);

end CONVERT;
```

Figure 6-8. Convert Generic

Despite how enclosed the abstraction is, in many ways, the limitations of the ADA language direct this form of construction. A language such as C++, with proper object inheritance would support these indirection routines much more easily.

6.6 Line Clipping

Drawing line objects within an X Window System environment is actually complicated by the limitations of the line clipping algorithms used. The problem manifests itself by saturation effects that arise because the maximum values that the X Window System supports is only 16 bits. With zooming facilities, it becomes very common to see lines do strange things within visuals. This combined with the fact that

filtering actions should be taken as early as possible in a graphics pipeline for increased performance, raised the requirement for a line clipping algorithm.

The classic line clipping algorithm, is that by Cohen-Sutherland [52], which relies on the assignment of what are termed outcodes in order to eliminate lines that fall outside a given region or completely inside the given region. This ensures that only lines that might need clipping are actually checked for intersection. Unfortunately, this algorithm sometimes performs needless clipping. The optimal algorithm developed to date has been the Nicholl-Lee-Nicholl algorithm. This algorithm minimises the number of intersection calculations. The net result is an algorithm that has three general cases defined by where a line segment starts relative to a region - in the centre, in a corner, or in an edge region as shown in figure 6-9..

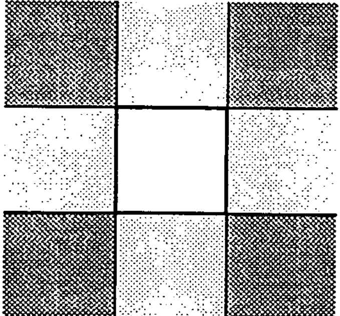


Figure 6-9. NCN Line Starting Points - Shown by Shading

The line clipping algorithm used in SAGE also minimises the number of intersection calculations, but instead defines six starting cases based on the start *and* end points of a line segment. The symmetries involved result in 81 (9x9) actual cases that have to be considered, of which 33 are the normal cases rejected by the equivalent of the Cohen-Sutherland outcode checks. By implementing the 81 choices as a case statement, a performance gain is attained. Figure 6-10 illustrates the six cases, and the names they

have been assigned, as well as the number of cases of the 81 possibilities that they map to.

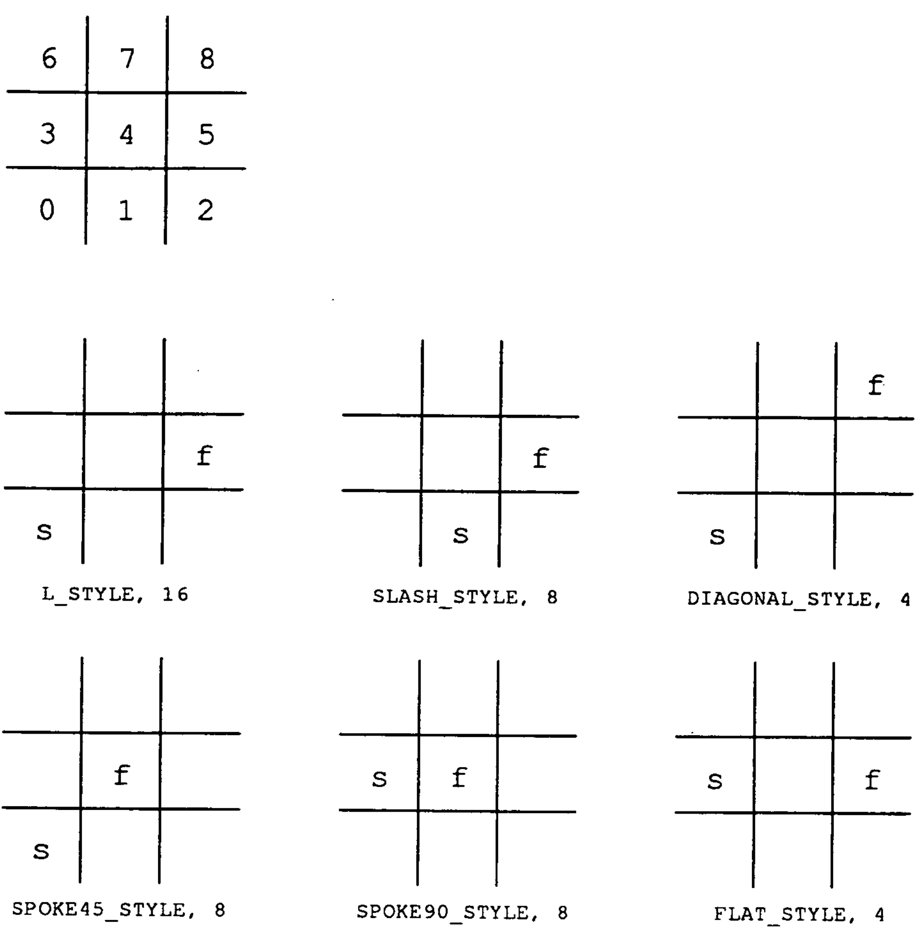


Figure 6-10. SAGE ***Start and End Line Points***

Rather than reproduce the entire case statement, which consists of several hundred lines, a representative example of the flat style code is shown in figure 6-11. Note the use of the 'T' value, which is equal to 9, to help demonstrate how the case value is computed.

```
--*------------------------------------------------------------------------*--
procedure CLIP_LINE(...) is
--*------------------------------------------------------------------------*--

--[------------------------------------------------------------------------[--
begin

  ...

  case PLACE is

    ...

    --!--------------------------------------------------------------------!--
    --!--------------------------------------------------------------------!--
    -- the FLAT_STYLE - 4

    --   | |    | |   |f|   |s|      6|7|8
    --  -----  -----  -----  -----    -----
    --  s| |f  f| |s   | |   | |      3|4|5
    --  -----  -----  -----  -----    -----
    --   | |    | |   |s|   |f|      0|1|2

    when   3 + T*5 => -- *frwd* normal
      FLAT_STYLE(X_1, Y_1, X_2, Y_2, U_X, U_Y, L_X, L_Y);
    when T*3 +   5 => -- *rvrs*
      FLAT_STYLE(X_2, Y_2, X_1, Y_1, U_X, U_Y, L_X, L_Y);

    when   1 + T*7 => -- *frwd*, swap the axis
      FLAT_STYLE(Y_1, X_1, Y_2, X_2, U_Y, U_X, L_Y, L_X);
    when T*1 +   7 => -- *rvrs*
      FLAT_STYLE(Y_2, X_2, Y_1, X_1, U_Y, U_X, L_Y, L_X);

    ....

  end case;

  ...

end CLIP_LINE;
--]------------------------------------------------------------------------]--
```

Figure 6-11. FLAT_STYLE ***Code Snippet For Line Clipping***

Since the vast majority of the line rejection is automatically handled by the rectangle manager, in many ways having an optimised line clipping algorithm is not as necessary as it may first appear. Nevertheless, since the requirement had to be satisfied, the development of the novel approach described above has been worthwhile.

'Twas the Night before Crisis

'Twas the night before crisis, and all through the house,
Not a program was working not even a browse.
The programmers were wrung out too mindless to care,
Knowing chances of cutover hadn't a prayer.
The users were nestled all snug in their beds,
While visions of inquiries danced in their heads.
When out in the lobby there arose such a clatter,
I sprang from my tube to see what was the matter.
And what to my wondering eyes should appear,
But a Super Programmer, oblivious to fear.
More rapid than eagles, his programs they came,
And he whistled and shouted and called them by name;
On Update! On Add! On Inquiry! On Delete!
On Batch Jobs! On Closing! On Functions Complete!
His eyes were glazed over, his fingers were lean,
From Weekends and nights in front of a screen.
A wink of his eye, and a twist of his head,
Soon gave me to know I had nothing to dread...

7. Results

The material in the preceding chapters has formed part of a real synthesis system that has been in used on numerous demonstrator designs. Although no silicon has actually been produced to date, the broad direction of the SAGE synthesis system, of making the designer an important part of the design loop has seen to be the correct approach. This chapter takes a step back from the SAGE system, and using material from numerous critical reviews [28, 29, 30], looks at what is good and bad about the SAGE synthesis system, particularly the aspects relating to the material explored in this thesis. To help place some of this analysis in context, the first section in this chapter begins by going through the design steps involved in a simple SAGE synthesis example with an aim to show how an electronic design engineer actually uses many of the tools that have been developed as part of the SAGE design methodology. This chapter completes by looking at the future work that could be carried out using the material presented in this thesis as a platform on which to build.

7.1 Design Example

The example that follows is the design of circuitry that can add three arbitrary numbers. The starting stage is the description of the problem in VHDL [31]. This is in two parts. Firstly the algorithmic specification of the problem, using the VHDL

package construct as shown in figure 7-1, and secondly the physical input and output behaviour specified by a process statement shown in figure 7-2. Whereas a more complex design can be composed of many arbitrary packages and procedural hierarchy, only one process statement with the instantiation of one of the procedures in the packages can be used.

```
package ADD_DEMO is

 type WORD is range -(2**15) to (2**15-1);

 procedure TWOADD(
  D1, D2, D3 : in WORD,
  SUM : out WORD
 );

end ADD_DEMO

package body ADD_DEMO is

 procedure TWOADD(
  D1, D2, D3 : in WORD,
  SUM : out WORD
 ) is
 begin
  SUM := D1+D2+D3;
 end TWOADD;

end ADD_DEMO;
```

Figure 7-1. VHDL ***Package Code***

The main part of the package consists of the type of arguments used and the actual computation contained within a subprogram. Note, since this procedure is used in the enclosing process statement, no use of a return statement is made, instead the result of the computation is passed back through last argument of the procedure called SUM.

```
use WORK.ADD_DEMO.all;

entity ADDER is
 port(
  INPUT : in WORD;
  OUTPUT: out WORD;
  CLOCK: in bit
 );
end ADDER

architecture CHIP of ADDER is
begin
 process
  variable D1, D2, D3 : WORD := 0;
  variable SUM : WORD := 0;
 begin
  D1 := INPUT;
  wait until CLOCK = '1';
  D2 := INPUT;
  wait until CLOCK = '1';
  D3 := INPUT;
  ADDER(D1, D2, D3, SUM);
  wait until CLOCK = '1';
  wait until CLOCK = '1';
  wait until CLOCK = '1';
  wait until CLOCK = '1';
  OUTPUT <= SUM;
  wait until CLOCK = '1';
 end process;
end CHIP;
```

Figure 7-2. VHDL ***Process Code***

The enclosing process statement is used to instantiate the algorithm to be synthesised, as well as define the required interface timing. Thus, in this example, all three input arguments arrive separated in time on one bus, while the output is produced on another bus. The timing is a specification of what a user desires, and may not actually be achieved by the synthesis process.

After a user has verified the operation of this code using normal simulation, this VHDL code is compiled using the VTIP [45] software and then translated into the SAGE

internal database format, namely BABBLE. At this stage, the SAGE synthesis tools can now be applied.

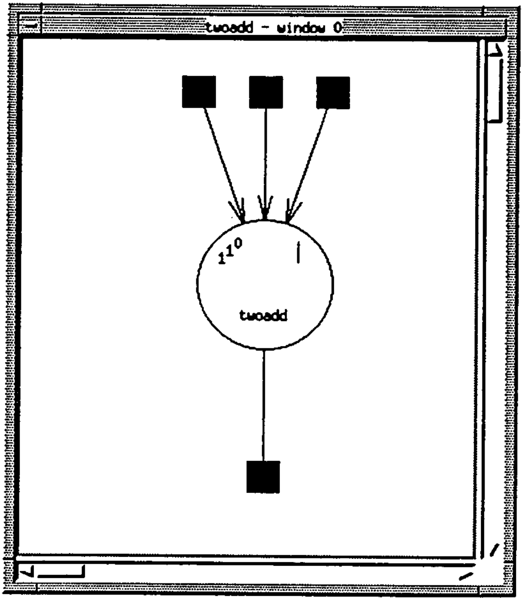
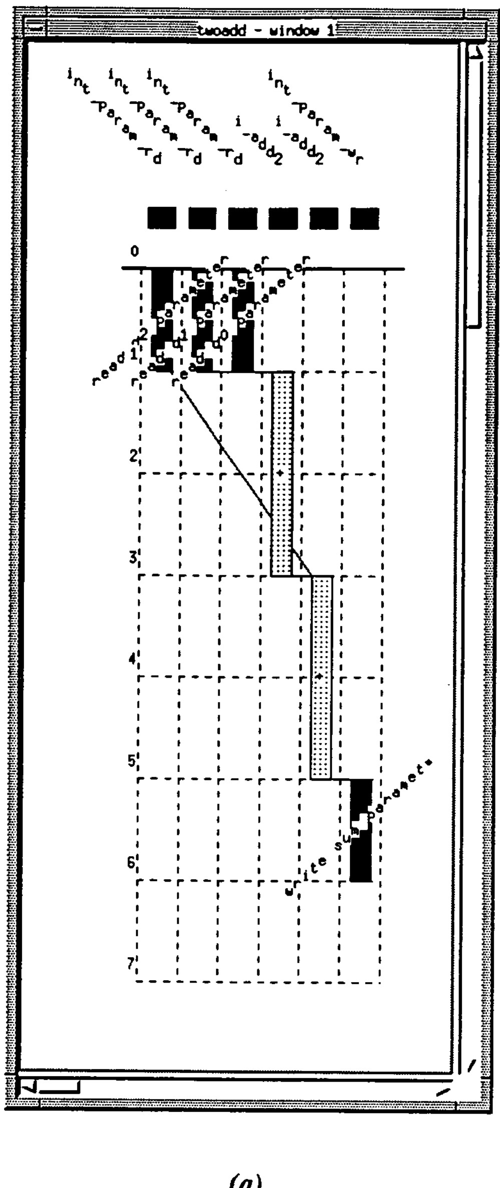


Figure 7-3. Control Flow for TWOADD Example

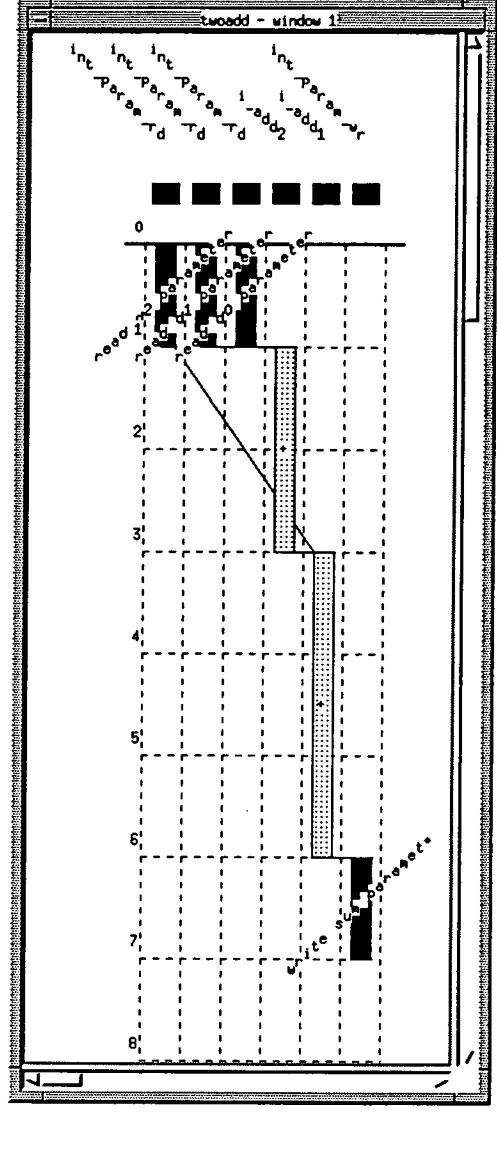
Once within the SAGE tool environment, a user is able to navigate around the key elements of the design. Since this example has no procedural hierarchy, there is only one design unit corresponding to the TWOADD subprogram. The control flow graph corresponding to the behaviour of this design unit is shown in figure 7-3. Its simplicity is a direct result of no control constructs such as 'for' loops being used in the TWOADD subprogram. The three input arcs represent the three numbers to be added, while the outbound arc is the result.

The figure represents a single basic block, whose internals corresponds to a single resource-time graph. The starting state of such a graph is illustrated in figure 7-4 (a). Here, the blocks along the top horizontal axis represent the allocation of resources, while the diagonal data flow represents the results of the initial 'as soon as possible' scheduling. The diagram has three parameter resources for supplying the three data values to be added and a single parameter resource to read the result. The corresponding structure graph consists of the two adder resources as shown in figure

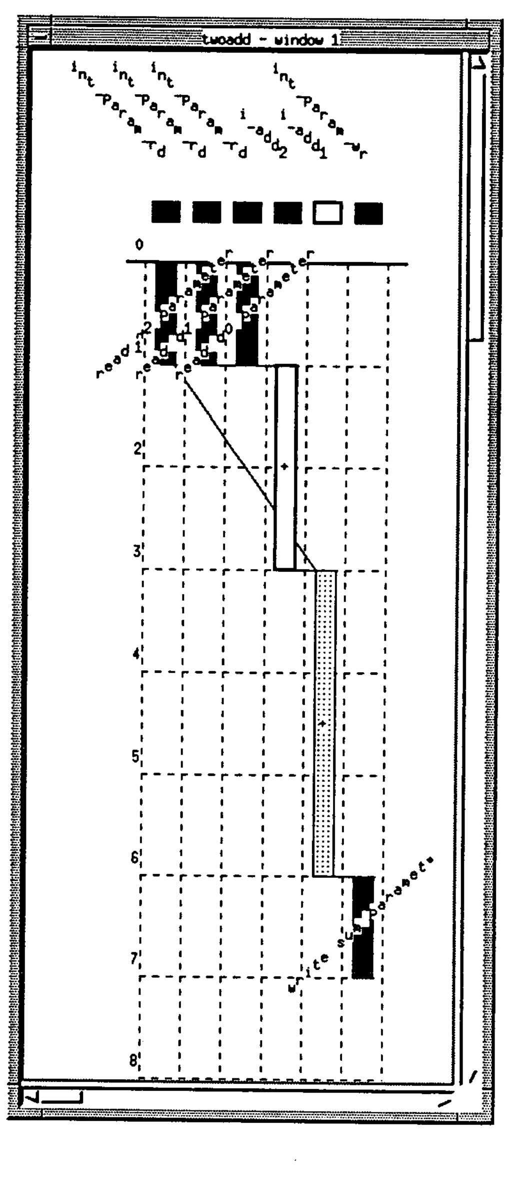
7-5 (a). The parameter resources correspond to the whiskers on the boundary of the structure graph.



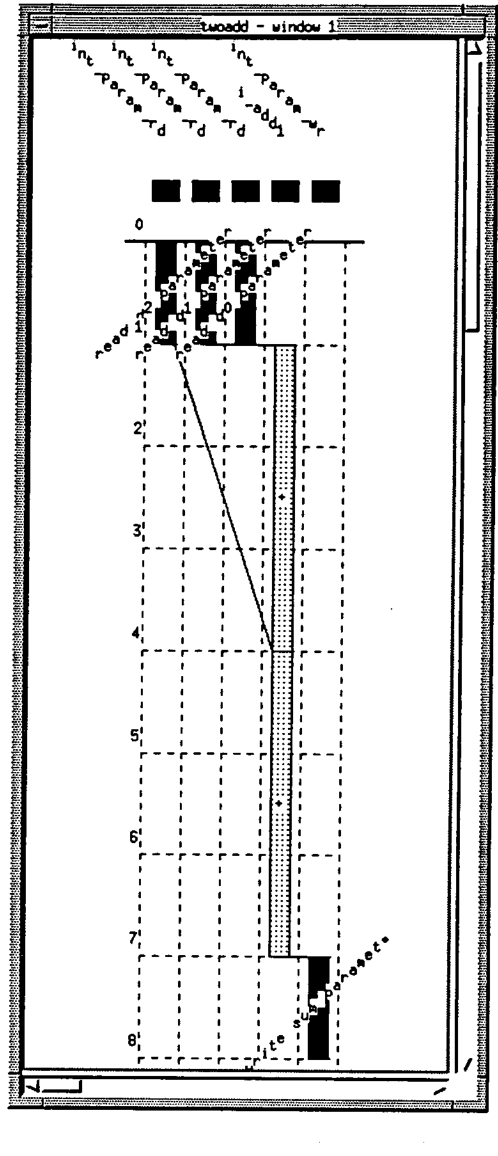
(a)



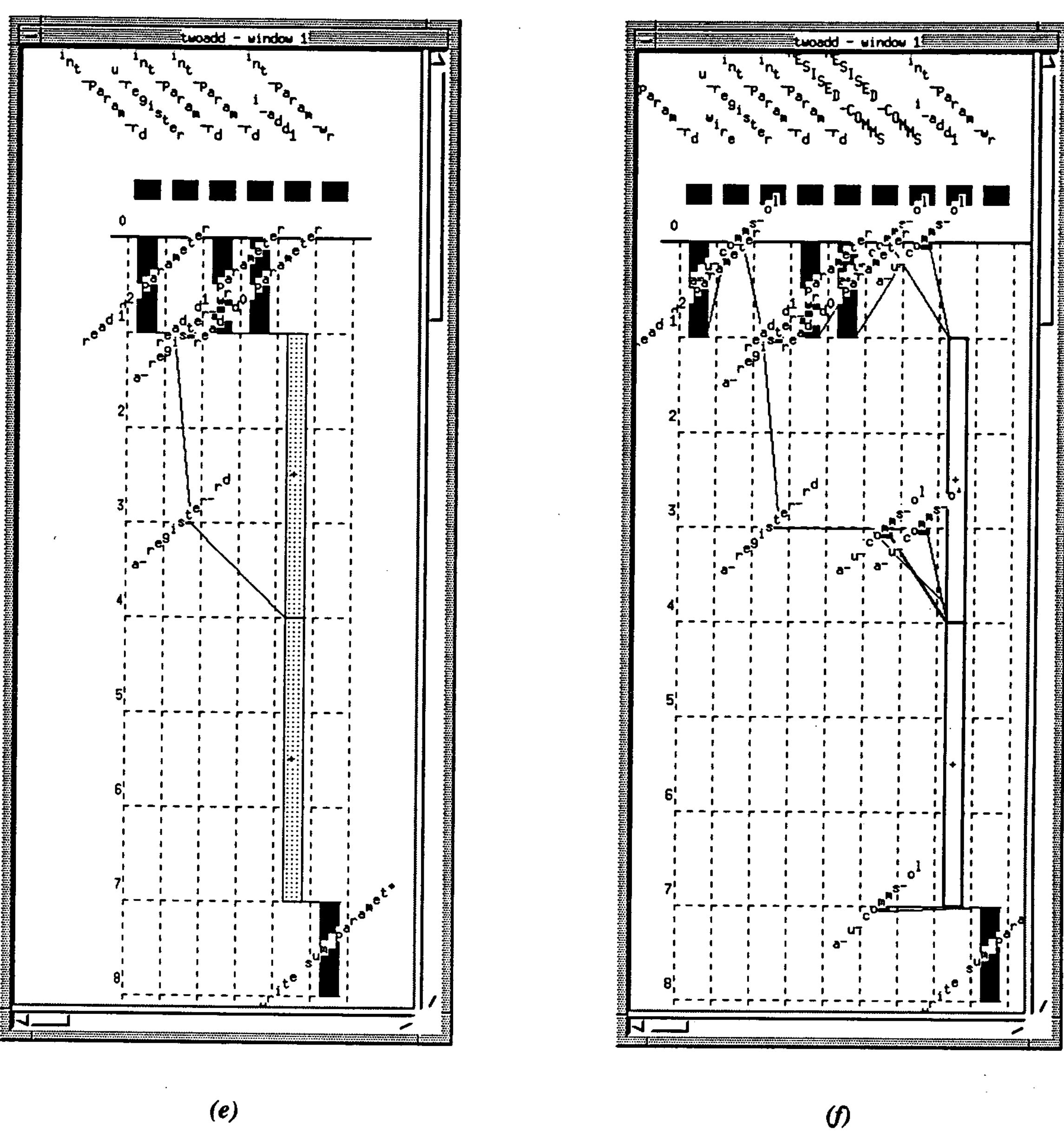
(b)



(c)



(d)



(e) *(f)*

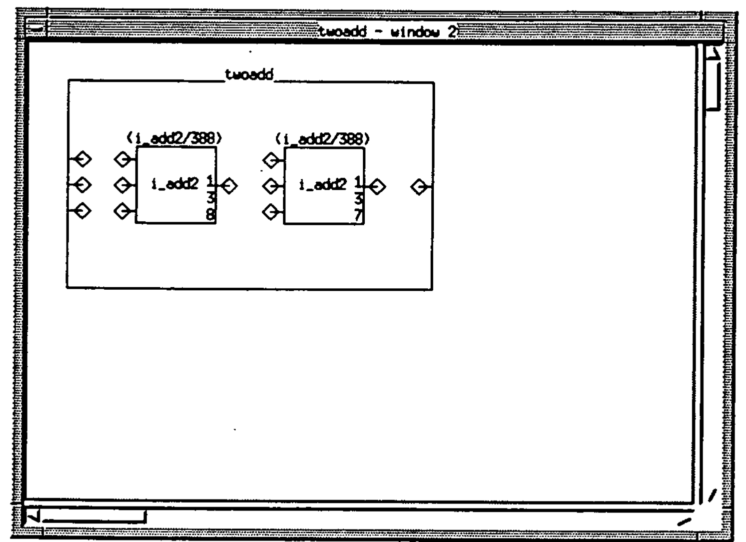
Figure 7-4. Resource-Time Graphs of* TWOADD *Example During Manipulation

The second resource-time graph (b), shows how the designer has selected a slower adder from the library for the second of the two add operations. In (c), the designer is ready to do a bind operation. The designer has selected the source call and the target resource, as shown by the highlighting. The result of this activity is shown in (d), where both additions now happen on the same resource. From a hardware point of view, this has impact on the routing of different information to the same ports on the adders. This is satisfied by the memory synthesis and communication synthesis steps

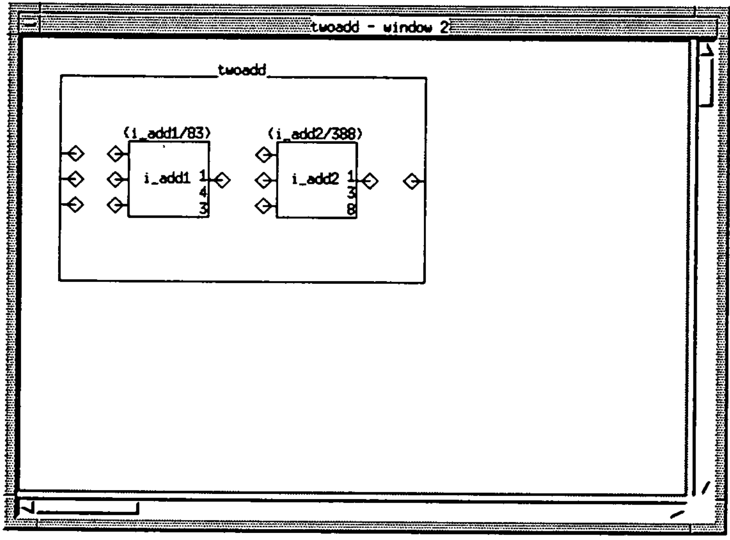
as highlighted by resource-time graphs (e) and (f) respectively. The newly created second resource on (e) shows how the first parameter is stored until required to be consumed. Although (f) is relatively complex, the main point to observe is the introduction of multiplexers to route the appropriate values to the input of the adders.

In step with the resource-time graphs, the structure graphs are also updated. In figure 7-5 (b), one of the adders `i_add2` has been replaced with the slower adder `i_add1`. By (c), the rebinding has happened, and the unused adder has been removed. The addition of extra hardware as shown in (d) and then (e), corresponds to the memory synthesis and communication synthesis steps respectively. The remaining unconnected pins on the adder and register corresponds to clock requirements, while the unconnected pins on the synthesised communications correspond to control requirements.

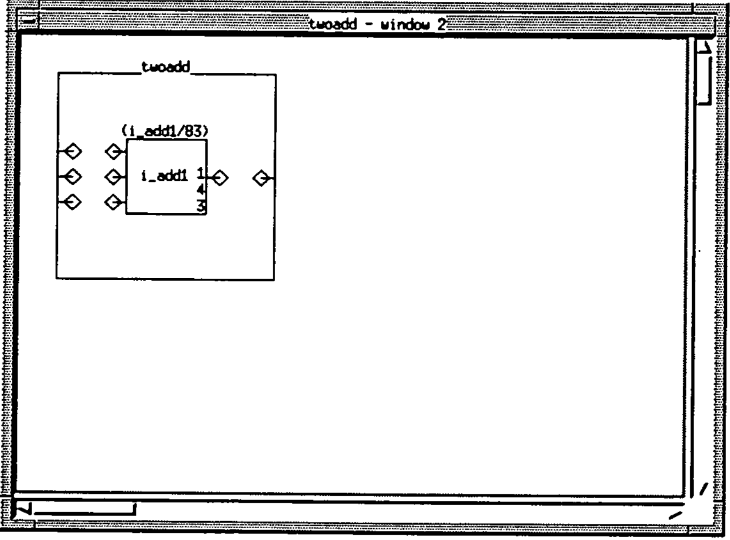
Formally, the next actions would be control synthesis, followed by netlist generation. If ram devices were used - i.e. greater than 1 register, then there is an additional requirement for address generation. Although interaction is allowed even after any synthesis stages, the tools will regress full design stages if design actions happen that upset the logical correctness of the database. Thus once a designer starts memory synthesis, all steps beyond this stage will generally clear out any created resources and implied data-flows if a user interacts with the resource-time graph. An example where this will not happen, is if a user only modifies register sharing.



(a)



(b)



(c)

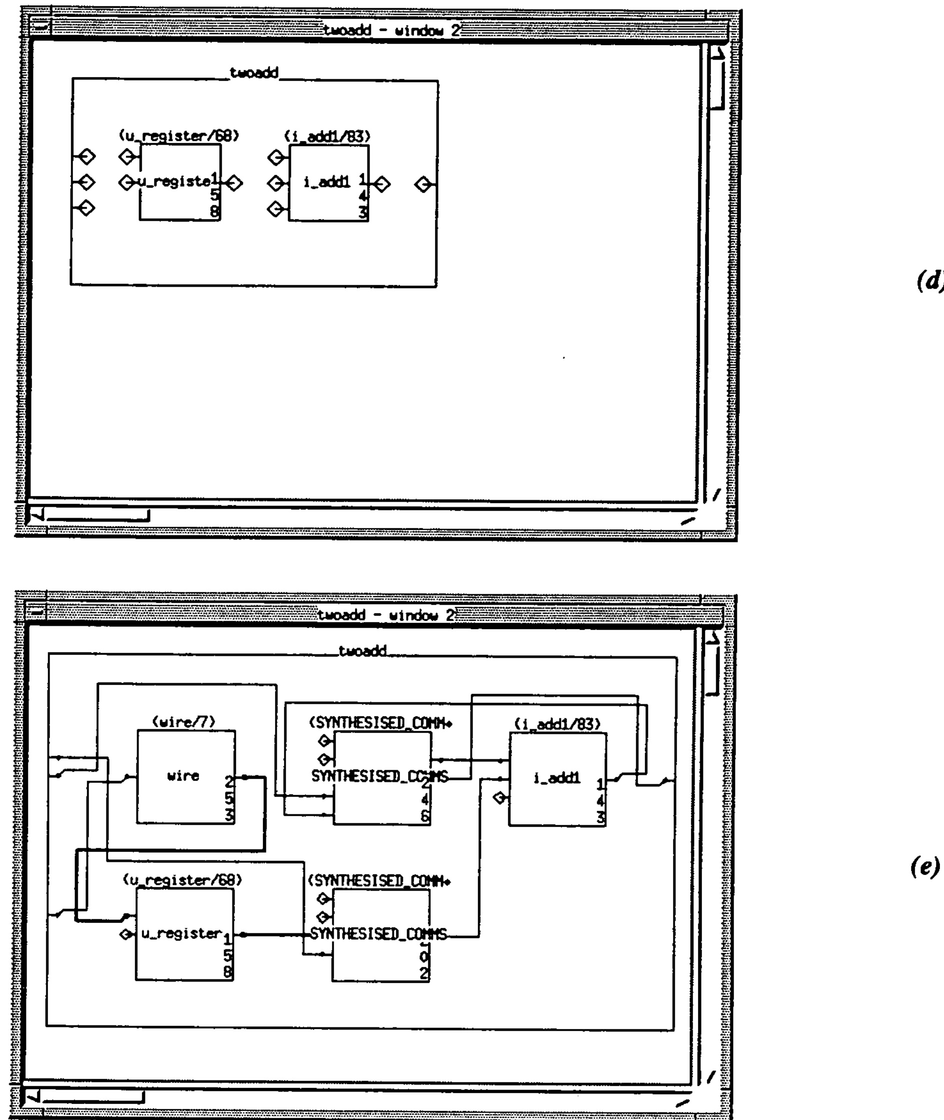


Figure 7-5. Structure Graphs of TWOADD *Example During Manipulation*

7.2 Critique

The acid test for the success of developed software, is the track record of usage after completion. In this regards, an objective observation would be that the SAGE toolset has been unsuccessful. More accurately, the project could be described as a 'success-disaster', in that there are many ideas that have been developed which have fulfilled the primary aims of the project even though as a whole, the software has not been applied in anger. As with many software based projects, when the target is clearer, a new software iteration will generally be faster to develop then continuing with the existing framework. In many ways this approach reflects the migration path from the early SAGE 2 software to SAGE 4, with the software being thrown away and the ideas carried across into the new tool generation. In many ways, this is the same path that the next generation of tools beyond SAGE 4 should follow.

At the heart of SAGE has been the need to support visibility of the inner workings of a digital system to help a designer guide the design process to a fine degree of granularity. In many respects and certainly at a superficial level, this has been successful. With the use of visuals, and particularly the resource-time graphs, a designer can see the two most important quantities, namely space and time, that will govern the performance of a system being implemented. Where these visuals start to become problematic is with large designs. There are two aspects. Firstly, within visuals, the information can become overwhelming even with the use of invisibility techniques to help manage information. Secondly, with large hierarchies, it becomes clear that the granularity at which design decisions can be made in a resource-time graph, can become irrelevant compared with the savings possible by manipulation at a higher level.

There are a multitude of inefficiencies that are covered by this second point. The most obvious example of this is highlighted by the process of mapping. If a mapped resource is used for a short period relative to the duration of the caller, it is made unavailable for the full length of the caller. Another relates to the way SAGE tools draw a clear and hard line between data path and control operations. In real designs, this is actually grey. Thus, with SAGE, a designer has no opportunity to manipulate

the control flow of a design except by rewriting the input VHDL. With the whole emphasis on trying to divorce specification from implementation, a large number of design activities could be defined as manipulating control flow structures and migrating data-flow between control structures as well as into and out of control structures. In many ways, the powerful ideas of retiming by the propagation of register elements could also apply. The third major impediment with the hierarchy model adopted in SAGE, is that of being rigid or inflexible. Although mapping in some ways diminishes the problem, it fails to recognise that specification hierarchy generally bears little relationship to the implementation hierarchy, and therefore the ability to dissolve given hierarchy and then, just as importantly if not more so, be able to recreate hierarchy, are both necessary to migrate a design conceptually and cleanly towards an efficient implementation. SAGE has the ability to dissolve hierarchy, but never to recreate it. In many ways, this last point is philosophical on the issue of whether the model represents behaviour with links to structure, or is a single model that can comfortably encompass both representations and therefore act as a medium through which the transformation form specification to implementation is able to happen. A fourth inefficiency relating to hierarchy is subtle, and relates to the way resource-time graphs are hierarchically composed of zones. For a given resource-time graph, only one zone can be computing at any one time, even if some zones are mutually exclusive in their operation. The core problem is that zones share the same hardware and therefore to prevent possible hardware resource clashes, it is an unfortunate but a necessary decision to say that only one zone per resource-time graph is active.

The resource-time graph is clearly very successful in the manipulation of data-flow activities to a very fine degree of granularity. Where its usefulness starts to breakdown is with operations that have indeterminate length, since it is no longer possible to use the space-time framework to mesh accurately together operations in a way that optimises a design. Another problem relates to the nature of the data flow operations themselves which have been collapsed into the same clock cycle. Such data flow corresponds to combinatorial logic and could in principle be decomposed using low-level logic synthesis algorithms. But, since this is a stage that has been designed to

happen at the next tool level that takes the netlist level output of SAGE, misleading results, in the form of being pessimistic could be obtained from the resource-time graphs. The classic example is that of unbalanced expressions, whose longest path will dictate the length of the computation, where an equivalent balanced expression through the application of simple algebraic rules could produce a significant temporal saving. This same form of manipulation applies at the clock step level, but issues concerning numerical accuracy can become the dominating effect as expression manipulation is made. The whole concept of using the resource-time graph breaks down further when trying to design complex functions that use mathematical operations like multiplication and division. The main problem is that the richness of choices in implementing such functions is not supported by the SAGE system. Although available devices can be encoded as a library device and made available through the matchmaking facilities, a designer has not the visibility of numerical accuracy issues or architectural issues as to choose a multiplier implementation from architectural choices that might range from simple add and shift operations, through to look up tables or beyond to full IEEE standards compliant floating point devices.

From a purely concept point of view, the resource-time graph can suggest to a novice designer that the aim of good design is to simply minimise resources and the length of time from input to output. Unfortunately, this hides one of the more powerful aspects of the resource-time graph, namely being able to control calculation throughput. The solution to this problem, would be an updated tessellation of a particular resource-time graph while a designer interacts with it, to show clearly how time is really composed of the two important quantities of throughput and latency.

Although providing a designer more information rather than less is a noble aim, there is a risk that unreasonable importance could be attached to the information made available. The major example here relates to the low level nano-second timing that a user has presented in a resource-time graph. With many of the current leading edge silicon technologies working at higher and higher clock speeds, the dominating influence is no longer just the computation time, but the communication time. Here, although the SAGE model in principle could be used in a back-annotation of layout delays mode, what would be more useful for a designer would be a forward estimation

model based on a characterisation of the target technology. If low-level nano-second timing is provided, it is fair to say that the normal asymmetric performance of silicon technologies should, with respect to rising and falling edge delays be reflected back in the resource-time graphs as an additional layer of information that could be available to a user.

Although SAGE supports fairly complex match making and unification facilities, in many ways they still fall far short of the way a designer would select key complex components. Whereas simple function blocks can be described, the heart of the matchmaking process relies on a simple name comparison. The remedy is the development of specialist matchers that focus on key technology areas, but the disadvantage is the specialism is difficult to capture and maintain. Another approach, is to let designers use their own ingenuity and break the correct by construction formality by allowing them to force selection of key components. Clearly, the vast majority of matchmaking and unification requirements can be simply be met by the existing facilities. The other main library modelling problem relates to the lack of parameterisation in the library cells.

Probably the biggest deficiency in the SAGE system, is that of completeness. In particular, the main missing stage to complete the path to implementation for any given design, is that of control generation. Since the manual generation of controllers is probably one of the harder synthesis steps, without any controllers being generated it would make practical application of SAGE on real designs next to impossible. The approach of having a controller for each element in the SAGE design hierarchy can also negate any performance gained by resource-time graph manipulation, because of the area and performance penalties that such controllers may impose. Another equally important element missing from SAGE is the need for fully characterised library elements, both for matchmaking within the SAGE system and post synthesis simulation to help confirm the validity of the implementation.

One of the main penalties imposed by the user interface is that of slow graph build times, which is compounded by the fact that such graphs need to be rebuilt after every successful interactive command. The other problem relates to the management of

windows during operation of the SAGE system. Since a large number of windows can be generated, facilities similar to those found in systems like Microsoft Windows 3.1, like automatic tiling and panelling are needed to help a users organise their workspace.

The general overall performance of the SAGE tools shows there is still a number of significant bottlenecks that should be eliminated to help man-machine interaction. In its present form, as well as the lack of maturity of the SAGE code, the lack of maturity of the tools on which SAGE is based on, also show through. As a medium level complexity example of a CORDIC algorithm consisting of 160 lines of C code shows, system resources on a typical SAGE session on a medium performance computer (circa 10 MIPS) takes over 25 MBytes of process space. This is in addition to 7 MBytes required by the X Window System. This same example taken from starting a SAGE session to completion using only the standard synthesis steps, takes over 20 minutes and produces output consisting of over 65000 lines of BABBLE code in 253 files occupying about 2.1 MBytes of disc space. Nearly two-thirds of this time is spent equally between the dummy controller insertion synthesis step and the save step.

With any complex tool, the problem of repeatability is very important, and is usually obtained by relying on a hosts computing resources. With the SAGE system, the problem is compounded by the fact that repeatability in the context of changes to the input VHDL source is required. Of the various evaluations of the SAGE system carried out, one of the main observations was the need to regularly change the input VHDL code to reflect the required architecture. Even if this was not the case, the issue of managing designer's mistakes in the specification would need to be handled. This is a difficult area, since in CAD tool development terms incremental change management is very difficult to implement. As has been seen, even the reverse incremental changes can be difficult to implement but is a necessity to support exploration of the design space.

7.3 Future Work

As the complexity of system requirements increase, as well as the CAD tools that are able to support their design, then the many levels of abstractions that exist between specification to implementation need to be brought closer together. The key point is the need to let a system designer work within one CAD environment and reach through all conceptual design levels as easily and swiftly as if the designer was working at one level. As an example, a designer could be specifying the general handshake requirements between two blocks in a petri-net transaction style of notation, but also be able to focus on the signals to specify required drive and low-level timing behaviour. Such an environment would have the necessary hooks to bring in important software development concepts that would allow complete system descriptions. In this way, a top-down design approach would allow the partitioning of a design at the hardware/software boundary, letting the time-critical aspects of a system be migrated towards a hardware implementation, while the software aspects migrate towards a suitable language with supporting microcontroller or microprocessor as appropriate. The work on CAD frameworks [64] which provide a common backbone to CAD tools is a suitable enabling technology on which such techniques can be developed from. Using the metaphor of the survival of the fittest, only the tools that provide the productivity benefits are kept, while tools that are below par can be evolved or replaced without having to start the development of the high-level synthesis system completely from scratch.

Although such a tighter integration of tools would place a requirement on designers to encompass a wider range of technologies, the extra understanding would be offset by the understanding embodied by the tools themselves. This is simply an extension of the idea that a designers ingenuity should be married efficiently and effectively with a range of computer based tools, with the demarcation and its purpose being clearly understood.

Although the SAGE graphics support have been designed to be optimal, there are a number of significant improvements that can be made. The main issue relates to incremental change of graphs. In the SAGE system, updated graphs were rebuilt for all

but the simplest of cases of highlighting actions. One of the main problems in rebuilding graphs, particularly relating to resource-time graphs, is that of moving large amounts of area from one part of a visual relative to another. The standard example is that of removing a column occupied by an unused resource, where all the columns to the right need to be moved leftwards. This could be overcome by an overlay mapping, that correlates different actual regions to where they are in the visual. This causes problems with objects that straddle the boundary. This concept could work hierarchically, as a picture gets more and more modifications done to it. Thus at some stage a decision must be made as to having to rebuild part or all of the graph that is being manipulated for efficiency reasons.

It is interesting to note that though resource-time graphs forms the main interface of SAGE, it is the broader manipulation of control flow graphs that can provide more significant gains in design optimisation. This is a pointer towards a new graphically based design approach. At the heart of this approach is the recognition that the simulation environment of current CAD systems as well as being the main proving ground for correctness of timing and functionality, could also be used as a digital system specification design tool, which in a generalised form could support synthesis through the use of behavioural transformations.

Such a tool would attack the problems of complex sequencing and attention to low-level timing details in one consistent environment. With time and signal space being the two most important quantities of concern to a designer, such a notation would let designers approach the problem from a blank sheet and in a top down or bottom fashion. With support of the three basic forms of control construct, namely sequencing, branching and looping, combined with hierarchy in time and signal space, such a tool would comfortably be able to bridge and manage the three major levels of digital level design, namely behaviour, register transfer and low-level timing.

The use of medium performance computers, selection of various programming standards and platforms, has in many ways affected the development of the SAGE system. With the continuing explosion in computing performance, and the tools that exploit such performance, the choices for tool designers become much richer. In

particular, with the aim being to provide as much information as possible to a designer to help them make an informed decision has to how to guide the design process, the technology of virtual reality will probably have the biggest impact on CAD design tools. At a more short term level, the use of more object orientated programming languages such as C++ and ADA9X will help the development time of high-level synthesis tools by helping CAD tool developers build more efficiently on existing software. Although the multi-process model has proved very successful for SAGE, the newer lightweight process models based on threads will help improve overall CAD tool performance.

The insight that graphics have provided to the SAGE system, has shown that their general application is a desirable goal. In this respect, the replacement of the front-end VHDL compilation engine with a graphical equivalent would improve the overall design flow. In many ways this move would mirror the path adopted in software development, where techniques over the years have migrated between textual and graphical based techniques as more powerful development tools have been developed [43]. For example, in the software realm, specification has been formalised into graphical techniques embodied in methodologies such as Yourdon. Similar approaches could be envisaged with hardware design specification, with the long term aim being that the starting stages for software and hardware specification will merge.

In many ways, retrospective application of the ideas presented in the foundations and framework chapters to the core synthesis tools which are supported by the SAGE system would produce significant improvements in efficiency. Additionally, these same foundations are sound enough to act as the basis on which to build a new generation of SAGE as well as tools for other parts of CAD requirements. Probably the most significant application outside the high-level synthesis requirements that could be made, is that of using the rectangle manager as a basis on which to develop an efficient high-performance routing engine.

The fascination with the bad points in a system, should not hide the fact that higher level tools can bring very many benefits if used in the correct fashion. Just as importantly, they can be the beacon pointing towards greater developments.

William Safire's Rules for Writers:

Remember to never split an infinitive. The passive voice should never be used. Do not put statements in the negative form. Verbs have to agree with their subjects. Proofread carefully to see if you words out. If you reread your work, you can find on rereading a great deal of repetition can be avoided by rereading and editing. A writer must not shift your point of view. And don't start a sentence with a conjunction. (Remember, too, a preposition is a terrible word to end a sentence with.) Don't overuse exclamation marks!! Place pronouns as close as possible, especially in long sentences, as of 10 or more words, to their antecedents. Writing carefully, dangling participles must be avoided. If any word is improper at the end of a sentence, a linking verb is. Take the bull by the hand and avoid mixing metaphors. Avoid trendy locutions that sound flaky. Everyone should be careful to use a singular pronoun with singular nouns in their writing. Always pick on the correct idiom. The adverb always follows the verb. Last but not least, avoid cliches like the plague; seek viable alternatives.

8. Conclusion

In the fast growing field of high-level synthesis, very little attention has been paid to the areas where core synthesis tools must interact with their immediate environment. Library modelling, netlist generation and design visualisation are the three interfaces that have been neglected at the expense of advances in core synthesis tools. This thesis has addressed this problem by looking at these primary interfaces and developing the ideas and tools that are needed to provide significant improvements over and above interfaces used by existing systems. The primary reason for this work, has been the need to help electronic design engineers become more productive by enabling them to handle the complexities of emerging design requirements.

Most of the results of this work have been embodied in the development of the SAGE high-level synthesis system, whose most significant difference between existing high-level synthesis systems is that the electronic design engineer is able to direct the process of synthesis to a very fine degree of granularity. The main vehicle that has helped achieve this is the visibility of design information through graphical representations with which a designer is able directly to interact. This is in stark contrast to the purely automatic approaches of many synthesis systems, whose only support in heading towards the desired solution tends to be in the form of restarting a synthesis session from scratch.

From the background review, the great variety and significant progress in the area of high-level synthesis systems can be seen. With the commercially driven CAD companies, even more significant progress has been made, in the form of tools that are

available in a fully supported form for electronic designer engineers to use in anger. With the research budgets of such organisations, the further improvement of these tools with enhancements that support many of the ideas attempted by the SAGE system, will ensure that electronic designers will have the tools to manage the complexities of system design problems of the near future.

Several new ideas and developments have been put forward in this thesis. This includes the development of library modelling language that can concisely capture the temporal behaviour of a wide variety of digital elements, and have the necessary selection functions to allow intelligent choice during the matchmaking and unification processes. On the purely structural side, a full exploration of all the problems in generating netlists with the development of the necessary concepts of creation, netlist attributing and name space handling for efficient and effective netlist generation has been presented. As well as a basic taxonomy of graphical representations for most aspects of digital system design representation, the key ideas of letting a designer graphically see the inner workings of a design and be able to guide the design process through the use of direct interaction has, as explained earlier, been one of the major areas explored by this thesis.

Many ideas and tools have been shown to be necessary to support the development of the interfaces. The framework concepts have been developed to marry efficiently the needs of the given systems building blocks with the needs of each of the high-level synthesis interfaces. The five key components are the multiple UNIX process models, the streamlined ADA to X language binding, development of the X Resources to support specific picture attributes across machine/screen/window tuples, a comprehensive rendering model and various concepts in user interface management services to help enhance man-machine interaction. The chapter on foundations has explored a range of packages. Though none are particularly special in their own right, as a collection they have in common flexibility, useability, simplicity, elegance and performance as their main contribution to the field.

For as many problems that have been addressed, there is an equal number of problems that have been deliberately avoided for pragmatic reasons. The most important of

these are issues relating to simulation, testability and design capture. Although these form rich areas for further exploration and extension of the SAGE system, there are a number of new avenues of exploration which have also been highlighted by the work in this thesis. The most interesting of these is the development of a design approach based on waveforms as commonly found in simulation engines. With the supporting concepts of abstraction in time and through the use of hierarchy, the vertical integration of the behaviour, register transfer and low level timing descriptions, a designer will be able to focus on the space-time behaviour of a design in a single consistent environment.

Whatever new tools are developed, the thirst for better, more capable and integrated tools will continue for the foreseeable future. In a handful of years, state of the art design systems will bear little relationship to current methodologies, not least of which the reasons will be the considerable advances in relatively inexpensive computing resources that the CAD tool developers will be able to exploit. But the getting there will be just as interesting as the journey. And in the case of the work presented in this thesis, has been.

References

1. Clocksin W.F., Mellish C.S., *"Programming in Prolog"*, Second Edition, Spinger-Verlag, 1984.

2. Aho A., Sethi R., Ullman J., *"Compilers: Principles, Techniques, Tools"*, Addison-Wesley, 1986.

3. Booch G., *"Software Engineering with ADA"*, Second Edition, Benjamin Cummings, 1987.

4. Young S.J., *"An Introduction to ADA"*, Second Revised Edition, Ellis Horwood, 1985.

5. Ichbiah J.D., et al., *"Reference Manual for the ADA Programming Language"*, ANSI/MIL-STD 1815 A, 1983.

6. Levine J.R, et al., *"Lex and Yacc"*, 2nd Edition, O Reilly and Associates, 1992.

7. Booch G., *"Software Components with Ada: Structures, Tools and Subsystems"*, Benjamin-Cummings, 1987.

8. Kernighan B., Ritchie D., *"The C Programming Language"*, ANSI C, Second Edition, Prentice Hall, 1988.

9. Stroustroup B., *"The C++ Programming Language"*, Addison-Wesley, 1991.

10. Winston P., Horn B., *"Lisp"*, Addison-Wesley, 1984.

11. Sedgewick R., *"Algorithms in C"*, Addison-Wesley, 1990.

12. Grant P.M., *"The DTI-Industry Sponsored Silicon Architectures Research Initiative"*, pp. 102-108, IEE Electronics and Communication Engineering Journal, June 1990.

13. Denyer P.B., et al., *"An Approach to the Synthesis of VLSI Systems from Behavioural Specifications"*, Philosopohy I, Project Document, December 1987.

14. Denyer P.B., et al., *"Architectural Synthesis with SARI"*, Philosopohy II, Project Document, May 1988.

15. Buchanan I., et al., *"SAGE 3.0 System Design"*, Project Document, December 1988.

16. Denyer P.B., *"SAGE Design Methodology"*, SAGE 2, SARI-035-D, Project Document, March 1989.

17. Denyer P.B., et al., *"SAGE System Architecture"*, SAGE 2, SARI-022-C, Project Document, May 1989.

18. Johal S.S., *"Resource Creation and Matching"*, SAGE 2, SARI-023-B, Project Document, April 1989.

19. Johal S.S., *"Netlist Generation"*, SAGE 2, SARI-024-B, Project Document, April 1989.

20. Johal S.S., *"Library Format"*, SAGE 2, SARI-020-B, Project Document, April 1989.

21. Johal S.S., *"Library Reference"*, SAGE 2, SARI-044-A, Project Document, February 1989.

22. Johal S.S., *"SAGE3 Graphical Interface"*, SARI-066-A, Project Document, June 1989.

23. Johal S.S., *"SAGE3 Design Visualisation"*, SARI-082-B, Project Document, September 1989.

24. Johal S.S., *"SAGE3 User Interface"*, SARI-083-B, Project Document, September 1989.

25. Johal S.S., *"The SAGE 3 Process Model"*, Project Technical Note, October 1989.

26. Johal S.S., *"Programmers Interface to the SAGE 3 User Interface"*, Project Technical Note, September 1989.

27. Johal S.S., *"ELLA Output to Display programs: elo2val, val2dpy, lut"*, GEC-Avionics, Internal Document, 1992.

28. Johal S.S., *"SAGE 4.2 Evaluation"*, GEC-Marconi Research Centre, Internal Document, Y/206/10648, November 1990.

29. Mallon D.J., *"SAGE 4.2 Plessey Evaluation Report"*, Internal Plessey Document, October 1990.

30. Denyer P.B., *"SAGE 4.2 Evaluation"*, Internal Document, October 1990.

31. Lipsett R., Intermetrics, et al., *"VHDL: Hardware Description and Design"*, Kluwer Academic Publishers, 1990.

32. Intermetrics, *"VHSIC Hardware Description Language"*, IEEE-1076, 1987.

33. EDIF Steering Committee, *"EDIF - Electronic Design Interchange Format Version 2 0 0"*, 1988.

34. ED Computer General, *"ELLA Language Reference Manual"*, Issue 5.1, 1986.

35. SPICE, *"SPICE 3f2, Users Manuel"*, University of California, Berkeley, 1992.

36. Cadence, *"Verilog-XL, Reference Manual, Volumes 1 and 2"*, Version 1.6, March 1991.

37. Mentor Graphics, *"IDEA Series BLM - Behavioural Level Modeling"*, March 1989.

38. Cadence, *"HDL Synthesizer and Optimizer, Modeling Style Guide for Mixed-Level Synthesis"*, Version 4.2, October 1991.

39. Computer General E.D., *"LOCAM Synthesis Toolset"*, 1991.

40. Mentor Graphics, *"Version 8, AUTOLOGIC"*, 1992.

41. Synopsis, "High-Level Design", Company Brouchure, HLD-DS 6/92-10M, 1992.

42. LSI Logic, *"Silicon 1076 - Advanced System Development Environment"*, Order No. 200036, November 1990.

43. i-Logix, *"No Need to Write: i-Logix Lets Designers 'Draw' VHDL"*, Military and Aerospace Electronics, Sentry Publishing, March 1991.

44. Mentor Graphics, *"Version 8, System Design Architect"*, 1992.

45. CLSI, *"VHDL Tool Integration Platform (VTIP)"*, DN0348-6, November 1990.

46. CLSI, *"Software Procedural Interface (SPI)"*, DN0263-13, November 1990.

47. CLSI, *"Design Library System (DLS)"*, DN0258-10, November 1990.

48. Mentor Graphics, *"QuickPart Modeling User's Manual"*, March 1989.

49. CMU, *"Serpent Overview"*, Carnegie Mellon University, SEI-89-UG-2, August 1989.

50. Rowe R., et al., *"The PICASSO Application Framework"*, University of Berkeley, September 1990.

51. SUN Microsystems, *"XView 1.0 Reference Manual: Summary of the XView API"*, August 1989.

52. Foley J., vanDam A., Feiner S., Hughs J., *"Computer Graphics, Principles and Practice"*, Addison-Wesley, 1990.

53. Samet H., *"Design and Analysis of Spatial Data Structures"*, Addison-Wesley, 1990.

54. Scheifler R., *"Xlib Protocol Reference Manual"*, Volume Zero, O Reilly & Associates, 1988.

55. Rye A., *"Xlib Programming Manual"*, Volume One, O Reilly & Associates, 1988.

56. O Reilly, *"Xlib Reference Manual"*, Volume Two, O Reilly & Associates, 1988.

57. Young D., *"The X Window System, Programming and Applications with Xt"*, OSF/Motif Edition, Prentice Hall, 1990.

58. Pitaksanonkul A. et al., *"Comparisons of Quad Trees and 4-D Trees: New Results"*, IEEE Transcations on Computer-Aided Design, pp. 1157-1164, Vol. 8, No. 11, November 1989.

59. Tamassia R., *"Automatic Graph Drawing and Readability of Diagrams"*, IEEE Transactions on Systems, Man and Cybernetics, pp 61-79, Vol. 18, No.1, February 1988.

60. Camposano R., *"From Behavior To Structure: High-Level Synthesis"*, pp. 8-19, IEEE Design & Test of Computers, October 1990.

61. McFarland M. C., Parker A. C., Camposano R., *"Tutorial on High-Level Synthesis"*, pp. 330-336, 25th ACM/IEEE Design Automation Conference, 1988.

62. Walker R.A., Camposano R., *"A Survey of High-Level Synthesis Systems"*, Kluwer Academic Publishers, 1991.

63. Fourman M.P., et al., *"Interactive Behavioural Synthesis"*, B5/1-13, Proceedings of the 1988 Electronic Design Automation Conference 1988.

64. CAD Framework Initiative, *"CFI Newsline"*, Vol. 2, No.1, February 1990.

65. Thomas D., et al., *"Linking the Behavioral and Structural Domains of Representation for Digital System Design"*, IEEE Transactions on Computer-Aided Design, pp. 103-110, Vol. CAD-6, No. 1, January 1987.

66. Tricky H., *"Flamel: A High-Level Hardware Compiler"*, IEEE Transactions on Computer-Aided Design, pp 259-269, Vol CAD-6, No. 2, March 1987.

67. Pangrle B.M., Gajski D., *" Design Tools for Intelligent Silicon Compilation"*, IEEE Transactions on Computer-Aided Design, pp. 1098-1112, Vol CAD-6, No. 6, November 1987.

68. Park N., Parker A., *"SEHWA: A Software Package for Synthesis of Pipelines from Behavioral Specifications"*, IEEE Transactions on Computer-Aided Design, pp. 356-370, Vol. 7, No. 3, March 1988.

69. Camposano R., Rosential W., *"Synthesing Circuits From Behavioral Descriptions"*, IEEE Transactions on Computer-Aided Design, pp. 171-180, Vol. 8, No. 2, February 1989.

70. Bushnell M.L., Director S.W., *"Automated Design Tool Execution in the Ulysses Design Enivronment"*, IEEE Transactions on Computer-Aided Design, pp. 279-287, Vol. 8, No. 3, March 1989.

71. Paulin P.G., Knight J.P., *"Force-Directed Scheduling for the Behavioral Synthesis of ASICs"*, IEEE Transaction on Computer-Aided Design, pp. 661-679, Vol. 8, No. 6, June 1989.

72. Andersson P., Philipson L., *"Movie - An Interactive Environment for Silicon Compilation Tools"*, IEEE Transactions on Computer-Aided Design, pp. 693-701, Vol. 8, No. 6, June 1989.

73. Walker R.A., Thomas D.E., *"Behavioral Transformation for Algorithmic Level IC Design"*, IEEE Transactions on Computer-Aided Design, pp. 1115-1128, Vol. 8, No. 10, October 1989.

74. Brewer F., Gajski D., *"CHIPPE: A System for Constraint Driven Behavioral Synthesis"*, IEEE Transactions on Computer-Aided Design, pp 681-695, Vol. 9, No. 7, July 1990.

75. McFarland M., Kowalski T., *"Incorporating Bottom-Up Design Into Hardware Synthesis"*, IEEE Transactions on Computer-Aided Design, pp. 938-950, Vol. 9, No. 7, September 1990.

76. Swinkels G.M., Hafer L., *"Schematic Generation with an Expert System"*, IEEE Transactions on Computer-Aided Design, Vol. 9, No. 12, December 1990.

77. Shung C.B., et al., *"An Integrated CAD System for Algorithm-Specific IC Design"*, IEEE Transactions on Computer-Aided Design, pp. 447-463, Vol. 10, No. 4, April 1991.

78. Knapp D.W., Parker A.C., *"The ADAM Design Planning Engine"*, IEEE Transactions on Computer-Aided Design, pp. 829-846, Vol. 10, No. 7, July 1991.

79. Bergamashi R.A., *"SKOL: A System for Logic Synthesis and Technology Mapping"*, IEEE Transactions on Computer-Aided Design, pp. 1342-1355, Vol. 10, No. 10, November 1991.

80. Bergamashi R.A., *"Synthesis of Concurrent VLSI Systems from Behavioural Descriptions"*, IEE Colloquium Digest on VLSI System Design: Specification and Synthesis, October 1987.

81. Kowalski T.J., *"The VLSI Design Automation Assistant: An Architecture Compiler"* Silicon Compilation, Addison-Wesley, pp. 122-152, 1988.

82. Brayton R.K., Camposano R., et al. *"The Yorktown Silicon Compiler"*, pp. 204-310, Silicon Compilation, Addison-Wesley, 1988.

83. Rabaey J., De Man H., et al., *"CATHEDRAL II: A Synthesis System for Multiprocessor DSP Systems"*, pp. 311-360, Silicon Compilation, Addison-Wesley, 1988.

84. Cheng E., Mazor S., *"The GENESIL Silicon Compiler"*, pp. 311-405, Silicon Compilation, Addison-Wesley, 1988.

85. Chu C-M. et al., *"HYPER: An Interactive Synthesis Environment for Real Time Applications"*, IEEE International Conference on Computer Design, pp. 432-435, October 1989.

86. Knapp D.W., *"Manual Rescheduling and Incremental Repair of Register Level Data Paths"*, IEEE International Conference on Computer-Aided Design, pp.58-61, November 1989.

87. Thomas D., *"The System Architects Workbench"*, Proc. of the 25th ACM/IEEE DAC, pp. 337-343, IEEE, 1988.

88. Devadas S., *"Algorithms for Hardware Allocation in Data Path Synthesis"*, IEEE Trans. on CAD, pp. 768-781, July 89.

89. Roy J., et al., *"DSS: A Distributed High-Level Synthesis System"*, IEEE Design & Test of Computers, pp. 18-31, June 1992.

90. Dutt N.D, *"Bridging High-Level Synthesis to RTL Technology Libraries"*, 28th ACM/IEEE Design Automation Conference, pp. 526-529, 1991.

91. May D., Keane C., *"Compiling OCCAM into Silicon"*, Communicating Process Architecture Document, Meiko, April 1986.

92. DeMicheli G., et al., *"The OLYMPUS Synthesis System"*, IEEE Design & Test of Computers, October 1990.

93. Lis J., et al., *"Synthesis from VHDL"*, IEEE International Conference on Computer Design, pp. 378-381, 1988.

94. Buset O.A., Elmasry M.I., *"ACE: A Hierarchical Graphical Interface for Architectural Synthesis"*, 26th ACM/IEEE Design Automation Conference, pp. 537-542, 1989.

Dissemination

As well as local GEC-Marconi Research Centre presentations etc., the following are public communications of work:

95. Denyer P.B., Johal S.S., *"Silicon Architectures Research Initiative"*, Poster Presentation, UK Design Automation Conference, 1989.

96. Johal S.S., Sardharwalla A.B., *"Interfacing GENESIL and SAGE"*, Poster Presentation, UK Design Automation Conference, 1989.

97. Johal S.S., Sardharwalla A.B., *"High Performance Graphics with X, Motif and ADA"*, Poster Presentation, UK Design Automation Conference, 1990.

98. Johal S.S., Denyer P.B., *"Motif, X And Ada In The Development Of SAGE: A Novel High-level Synthesis System"*, pp. 78-87, European X Window System Conference, November 1990.

```
--$-------------------------------------------------------------------------------------------$--
--$  Features   :  o(1)  for ADD, DEL to the end or start of the list                          $--
--$                 o(1)  for GETs in a sequential order - either forward or backward          $--
--$                 o(1)  operation for SIZE and REVERSE operations                            $--
--$                 o(n/2) to find items with random seeks                                     $--
--$                 Full consistency is maintained with multiple uses of the same list object (eg. if  $--
--$                 if A and B point to the same list, and we add a new element to the front of B,   $--
--$                 then A will also see the new element - note, Booch lists don't handle this case)  $--
--$                                                                                            $--
--$  Problems   :  ALSYS ADA does not like any local generic templates to be locally stamped out.  $--
--$                 Thus, a couple of these generics have been wrapped up with a procedural shell.  $--
--$-------------------------------------------------------------------------------------------$--

with MISC; -- only FREE_TEMPLATE is used from in here
generic

  type ITEM_T is private;

  -- Since the compiler can be used to control type checking, we can specifiy the type of INTEGERS used.
  -- In some cases, strong type checking is a nuisance, whence INTEGER_T can be tied to say LONG_INTEGER.
  -- In other cases, it is a useful mechanism to ensure strong type checking/consistency. Ensure that the
  -- used type extends from at least -1 to a largish positive value as a minimum range.
  type INTEGER_T is range <>;

  -- just in case the check for equality is a bit complicated
  with function EQUAL(A, B : ITEM_T) return BOOLEAN is "=";

--g-------------------------------------------------------------------------------------------g--
package DLISTS is
--g-------------------------------------------------------------------------------------------g--

  -- All lists are objects. Once created, their constitient elements are counted from 1.
  -- Since lists are objects, they *need* explicit destruction, thru' use of one of the DESTROY procedures

  -- exception raised (usually) when 'not INDEX in 1..SIZE'
  DLISTS_CONSTRAINT_ERROR : exception;

  type LIST_T is private;

  function INIT return LIST_T;
  function VALID(LIST : in LIST_T) return BOOLEAN; -- tells if the LIST is initialised
```

```
procedure DUMMY_FREE is new MISC.FREE_TEMPLATE(ITEM_T); -- used as default in some generics

generic
  with procedure DESTROY(ITEM : in out ITEM_T);
procedure DESTROY_TEMPLATE(THIS : in out LIST_T);

-- 'procedure DESTROY is new DESTROY_TEMPLATE(DUMMY_FREE);', prevented because of an ALSYS bug
procedure DESTROY(THIS : in out LIST_T);

function SIZE(LIST : in LIST_T) return INTEGER_T;

--!--------------------------------------------------------------------------------------------!--
-- the following 'ADD' elements according to the 'WHERE' value
-- 0 - make it 1st, 1 - make it 2nd etc., -1 - add it to the end, other - raise an exception
procedure ADD (TO_THIS : in LIST_T; THIS : in ITEM_T; WHERE : in INTEGER_T := -1);
-- addition of one list to another, destroying the list being added
procedure ADD (TO_THIS : in LIST_T; THIS : in out LIST_T; WHERE : in INTEGER_T := -1);

function GET(LIST : in LIST_T; NO : in INTEGER_T) return ITEM_T;

generic
  with procedure DESTROY(ITEM : in out ITEM_T);
procedure DEL_TEMPLATE(FROM_THIS : in LIST_T; THIS : INTEGER_T);

-- 'procedure DEL is new DEL_TEMPLATE(DUMMY_FREE);', prevented because of an ALSYS bug
procedure DEL(FROM_THIS : in LIST_T; THIS : INTEGER_T);

procedure REPLACE(IN_LIST : in LIST_T; WITH_THIS : in ITEM_T; WHERE : in INTEGER_T);

generic
  with function COPY(THIS : ITEM_T) return ITEM_T;
function COPY_TEMPLATE(THIS : LIST_T) return LIST_T;

-- 'function COPY is new COPY_TEMPLATE(DUMMY_COPY)', prevented because of an ALSYS bug
function COPY(THIS : LIST_T) return LIST_T;

procedure REVRSE(LIST : LIST_T);
```

```
--!----------------------------------------------------------------------------------------!--
-- (1) DEL_DUPL_0: remove all items of a given type, when more than one is encountered, and
-- (2) DEL_DUPL_1: to remove the duplicates (ie. if n identical items are encountered, 1 is always left).

-- (1)
generic
  with function MATCH(A, B : ITEM_T) return BOOLEAN is EQUAL;
  with procedure DESTROY(ITEM : in out ITEM_T) is DUMMY_FREE;
procedure DEL_DUPL_0_TEMPLATE(IN_THIS : in LIST_T);

-- (2)
generic
  with function MATCH(A, B : ITEM_T) return BOOLEAN is EQUAL;
  with procedure DESTROY(ITEM : in out ITEM_T) is DUMMY_FREE;
procedure DEL_DUPL_1_TEMPLATE(IN_THIS : in LIST_T);

generic
  with function MATCH(A, B : ITEM_T) return BOOLEAN is EQUAL;
function FIND_TEMPLATE(LIST : LIST_T; THIS : ITEM_T) return INTEGER_T;

generic
  with function MATCH(A, B : ITEM_T) return BOOLEAN is EQUAL;
function IS_MEMBER_TEMPLATE(LIST : LIST_T; THIS : ITEM_T) return BOOLEAN;


-- The above routines can be stamped out as shown below - but ALSYS ada has problems

-- procedure DEL_DUPL is new DEL_DUPL_TEMPLATE;
-- function FIND is new FIND_TEMPLATE;
-- function IS_MEMBER is new IS_MEMBER_TEMPLATE;

private

   type LIST_REC_T;
   type LIST_T is access LIST_REC_T;

end DLISTS;
```

Figure A-1. The Lists Generic

```
with DLISTS, TEXT_IO, MISC;
generic

  type KEY_T is private;
  type DATA_T is private;

  with function "<"(X, Y : KEY_T) return BOOLEAN is <>;
  with function ">"(X, Y : KEY_T) return BOOLEAN is <>;

--g-------------------------------------------------------------------------------------------------g--
package RED_BLACK_TREES is
--g-------------------------------------------------------------------------------------------------g--

  -- general RED-BLACK tree implementation for log(n) search and add.
  -- see SEDGWICK, Chapter 15, Algorithms, 2nd Edition, Addison-Wesley.

  -- *warning*, this package provides generics, such that constructors and
  -- destructors (ADD/DEL) must not be used with the passed subprograms
  -- the ADD/DEL routines do *not* check that they are in a valid context
  -- to make ammends, on of the generic allows elements to be deleted

  -- only the 'fn GET(TREE,KEY) return DATA_T' and 'proc DEL(TREE,KEY)' raise this exception
  RED_BLACK_KEY_NOT_FOUND : exception;

  --!-----------------------------------------------------------------------------------------------!--

  type TREE_T is private;

  -- rather than require the user to provide 'sentinal' edge values, we use this:
  type SEARCH_KEY_T(IS_KEY : BOOLEAN := TRUE) is record
    case IS_KEY is
      when TRUE => KEY : KEY_T;
      when FALSE => null; -- means an edge value
    end case;
  end record;

  type DIRECTION_T is (LEFT_RIGHT, RIGHT_LEFT);

  EDGE : SEARCH_KEY_T(IS_KEY => FALSE); -- this is a constant
```

```
--!-------------------------------------------------------------------------------------------!--
-- the following is the list of options for the searches and deletes possible
-- note, the iteration is inclusive. (the squiggles below are the search 'edges')
--
--      LEFT_RIGHT                         RIGHT_LEFT
--
--   \                                 \
--   / ----------> V                   / <---------- V
--   \                                 \
--
--                 \                                 \
--   V ----------> /                   V <---------- /
--                 \                                 \
--
--   \             \                   \             \
--   / ----------> /                   / <---------- /
--   \             \                   \             \
--
--      s       f                         s       f
--     V------->V                        V<-------V        START.KEY <= FINISH.KEY
--
--
--   \     f   s   \                   \     f   s   \
--   /--->V   V--->/                   /<---V   V<---/     START.KEY >  FINISH.KEY
--   \             \                   \             \

package DATA_LST_PAC is new DLISTS(DATA_T, LONG_INTEGER);
subtype DATA_LST_T is DATA_LST_PAC.LIST_T;
use DATA_LST_PAC;

function INIT return TREE_T;

function VALID(TREE : TREE_T) return BOOLEAN;
```

```
--!----------------------------------------------------------------------------------------------!--

procedure ADD(TO_THIS : TREE_T; KEY : KEY_T; THIS_DATA : DATA_T);

-- normal get, will return a single data item, even if there are repeated keys otherwise an exception
function GET(TREE : TREE_T; KEY : KEY_T) return DATA_T;
-- with repeated keys, this GET might can return zero or more data items
function GET(TREE : TREE_T; KEY : KEY_T) return DATA_LST_T;
-- get all the items within a 'range' over the given keys, again returning zero or more data items
function GET(TREE : TREE_T; KEY_START, KEY_FINISH : SEARCH_KEY_T; DIRECTION : DIRECTION_T := LEFT_RIGHT)
return DATA_LST_T;

-- note, in the last two 'GET' fns above, explicit storage reclaming is necessary for DATA_LST_T

--!----------------------------------------------------------------------------------------------!--

function SIZE(TREE : TREE_T) return LONG_INTEGER;

function EXISTS(TREE : TREE_T; KEY : KEY_T) return BOOLEAN;

--!----------------------------------------------------------------------------------------------!--
generic
  with function COPY(DATA : DATA_T) return DATA_T;
function COPY_TEMPLATE(TREE : TREE_T) return TREE_T;

function COPY(TREE : TREE_T) return TREE_T;

--!----------------------------------------------------------------------------------------------!--
-- destroy the tree object, including any contents
generic
  with procedure DESTROY(DATA : in out DATA_T);
procedure DESTROY_TEMPLATE(TREE : in out TREE_T);

procedure DESTROY(TREE : in out TREE_T);

--!----------------------------------------------------------------------------------------------!--
-- delete one item, with the given key value, raise an exception if key is not found
generic
  with procedure DESTROY(DATA : in out DATA_T);
procedure DEL_TEMPLATE(TREE : in TREE_T; KEY : KEY_T);
procedure DEL(TREE : in TREE_T; KEY : KEY_T);
```

```
--!-----------------------------------------------------------------------------------------!--

-- general purpose iterators (as opposed to the dedicated GET routines)
generic
  with function REGISTER(DATA : DATA_T) return BOOLEAN; -- if result false, we stop iterating
procedure ITERATE(TREE : TREE_T; START, FINISH : SEARCH_KEY_T; DIRECTION : DIRECTION_T);

-- this generic provides 'all purpose' functionality - some of which *can* be abused. facilities are:
--   (1) DATA and KEY can be directly modified.
--   (2) The DATA/KEY pairs can be deleted during the iteration
-- KEY changes should be done very carefully, ensuring that the resulting TREE will have
-- an unchanged iteration order.
generic
  with procedure REGISTER(
    DATA : in out DATA_T; KEY : in out KEY_T; DEL : out BOOLEAN; CONTINE : out BOOLEAN
  );
  -- if DEL is true, the referenced item is destroyed
  -- (and then) if CONTINUE is false, we stop iterating
procedure ITERATE_MODIFY(TREE : TREE_T; START, FINISH : SEARCH_KEY_T; DIRECTION : DIRECTION_T);

--!-----------------------------------------------------------------------------------------!--

-- this is for testing purposes - a full explanation is in the body
-- part associated with this spec - needless to says, its wild !
generic
  with function DISPLAY(DATA : DATA_T) return STRING;
  with function DISPLAY(KEY : KEY_T) return STRING;
procedure DUMP(TREE : TREE_T; SPLIT : INTEGER; CHANNEL : TEXT_IO.FILE_TYPE := TEXT_IO.STANDARD_OUTPUT);

private

  type TREE_REC_T;
  type TREE_T is access TREE_REC_T;

end RED_BLACK_TREES;
```

Figure A-2. Red-Black Trees Generic

```
generic

  type KEY_T is private;
  type DATA_T is private;

  with function "<"(X, Y : KEY_T) return BOOLEAN;
  with function ">"(X, Y : KEY_T) return BOOLEAN;

--g-----------------------------------------------------------------------------------------g--
package BTREES is
--g-----------------------------------------------------------------------------------------g--

  BTREES_KEY_ALREADY_EXISTS,
  BTREES_KEY_NOT_FOUND,
  BTREES_NEXT_KEY_NOT_FOUND : exception;

  type BINARY_T is private;
  type SEARCH_STATE_T is limited private; -- used for searches

  procedure PUT_DATA(BINARY : in out BINARY_T; KEY : KEY_T; DATA : DATA_T);
  function GET_DATA(BINARY : BINARY_T; KEY : KEY_T) return DATA_T;

  function EXISTS(BINARY : BINARY_T; KEY : KEY_T) return BOOLEAN;

  function COPY(BINARY : BINARY_T) return BINARY_T;

  procedure DELETE(BINARY : in out BINARY_T);
  procedure DELETE(BINARY : in out BINARY_T; KEY : KEY_T);

  procedure PRE_ORDER_START(BINARY : BINARY_T; SEARCH_STATE : out SEARCH_STATE_T);
  procedure IN_ORDER_START(BINARY : BINARY_T; SEARCH_STATE : out SEARCH_STATE_T);
  procedure POST_ORDER_START(BINARY : BINARY_T; SEARCH_STATE : out SEARCH_STATE_T);

  procedure PRE_ORDER_NEXT(SEARCH_STATE : in out SEARCH_STATE_T; KEY : out KEY_T; DATA : out DATA_T);
  procedure IN_ORDER_NEXT(SEARCH_STATE : in out SEARCH_STATE_T; KEY : out KEY_T; DATA : out DATA_T);
  procedure POST_ORDER_NEXT(SEARCH_STATE : in out SEARCH_STATE_T; KEY : out KEY_T; DATA : out DATA_T);

  procedure START(BINARY : BINARY_T; KEY : KEY_T; SEARCH_STATE : in out SEARCH_STATE_T);

  procedure LAST(SEARCH_STATE : SEARCH_STATE_T; LAST : out BOOLEAN);
```

```
private

  type SEARCH_STATE_T is
    record                  -- limited, 'cause its internal to the trees
      STATE : BINARY_T;     -- in a record, 'cause ADA will not have it any other way
    end record;             -- ('subtype ...', '... is BINARY_T' and renames do not work)

  type BINARY_REC_T;
  type BINARY_T is access BINARY_REC_T;
  type BINARY_REC_T is
    record
      UP : BINARY_T;
      LEFT, RIGHT : BINARY_T;
      KEY : KEY_T;
      DATA : DATA_T;
    end record;

end BTREES;
```

Figure A-3. Binary Trees Generic

```
with LOOKUP;
generic

  type KEY_ELEMENT_T is private;
  type KEY_T is array(POSITIVE range <>) of KEY_ELEMENT_T;
  type KEY_PTR_T is access KEY_T; -- used by search routines, since they can't be fns
                                  -- they are unable to return a variable array
  type DATA_T is private;

  with function "<"(X, Y : KEY_ELEMENT_T) return BOOLEAN;
  with function ">"(X, Y : KEY_ELEMENT_T) return BOOLEAN;

--g---------------------------------------------------------------------------------g--
package TREES is
--g---------------------------------------------------------------------------------g--

  TREES_KEY_ALREADY_EXISTS,
  TREES_KEY_NOT_FOUND,
  TREES_INDEX_NOT_FOUND,
  TREES_NEXT_KEY_NOT_FOUND : exception;

  type TREE_T is private;
  type SEARCH_STATE_T is limited private;

  procedure PUT_DATA(TREE : in out TREE_T; KEY : KEY_T; DATA : DATA_T);
  function GET_DATA(TREE : TREE_T; KEY : KEY_T) return DATA_T;
  function GET_TREE(TREE : TREE_T; KEY : KEY_T) return TREE_T;

  function EXISTS(TREE : TREE_T; KEY : KEY_T) return BOOLEAN;

  function COPY(TREE : TREE_T) return TREE_T;

  procedure DELETE(TREE : in out TREE_T);
  procedure DELETE(TREE : in out TREE_T; KEY : KEY_T);

  procedure START_BY_LEVEL(TREE : TREE_T; SEARCH_STATE : in out SEARCH_STATE_T);
  procedure START_BY_DEPTH(TREE : TREE_T; SEARCH_STATE : in out SEARCH_STATE_T);

  procedure NEXT_BY_LEVEL(
    SEARCH_STATE : in out SEARCH_STATE_T; KEY_PTR : out KEY_PTR_T; DATA : out DATA_T
  );
```

```
  procedure NEXT_BY_DEPTH(
    SEARCH_STATE : in out SEARCH_STATE_T; KEY_PTR : out KEY_PTR_T; DATA : out DATA_T
  );

  function LAST(SEARCH_STATE : SEARCH_STATE_T) return BOOLEAN;

  function COUNT(TREE : TREE_T) return NATURAL;
  function GET_DATA(TREE : TREE_T; INDEX : NATURAL) return DATA_T;
  function GET_INDEX(TREE : TREE_T; KEY_ELEMENT : KEY_ELEMENT_T) return NATURAL;

private

  type TREE_REC_T(IS_ROOT : BOOLEAN := FALSE);
  type TREE_T is access TREE_REC_T;

  type DATA_RECORD_T is
    record
      DATA : DATA_T; -- necessary, cause no equiv of 'C' & op
    end record;

  ...

end TREES;
```

Figure A-4. nary Trees Generic

```
generic
   --!---------------------------------------------------------------------------------------!--

   type DATA_T is private;

   type COORD_T is private;                                    -- can be FLOAT, INTEGER, LONG_INTEGER etc.
   type COORD_INDEX_T is range <>;                             -- will be 1..4 only
   type COORD_ARR_T is array(COORD_INDEX_T) of COORD_T;        -- read as, (x,y,X,Y)

   with function SIZE(DATA : DATA_T) return COORD_ARR_T;

   --!---------------------------------------------------------------------------------------!--

   --                                     examples, with COORD_T being FLOAT:

   MINIMUM :  COORD_T;                    -- COORD_T'SAFE_SMALL*100.0
   ZERO :  COORD_T;                       -- 0.0
   MINUS_ONE :  COORD_T;                  -- -1.0
   TWO :  COORD_T;                        -- 2.0
   DEFAULT_BOUNDS :  COORD_ARR_T;         -- (-10.0, -10.0, 10.0, 10.0)

   --!---------------------------------------------------------------------------------------!--

   -- for internal bound expansion, and to prevent NUMERIC_ERRORs, we need to convert
   -- between COORDs and LONG_FLOATs. (this is not possible internally, because COORD_T is private)
   with function COORD2LONG_FLOAT(COORD : COORD_T) return LONG_FLOAT;

   --!---------------------------------------------------------------------------------------!--

   -- *
   -- all the following can be LEFT OUT during instantiation, 'cause they have defaults
   -- *

   -- when building the quad tree, this figure determines when a region is divided.
   -- it can be overidden in the initialisation routine called 'INIT'.
   ADD_QUAD_TREE_THRESHOLD_DIVIDE_FIGURE : LONG_INTEGER := 3;

   -- in order to check the validity of rectangles during deletion, the following is used.
   with function EQUAL(A, B : DATA_T) return BOOLEAN is "=";
```

```
-- the numerical COORD_T must have the following operations avialable ...
with function ">"(A, B : COORD_T) return BOOLEAN is <>;
with function "<"(A, B : COORD_T) return BOOLEAN is <>;
with function ">="(A, B : COORD_T) return BOOLEAN is <>;
with function "<="(A, B : COORD_T) return BOOLEAN is <>;

with function "+"(A, B : COORD_T) return COORD_T is <>;
with function "-"(A, B : COORD_T) return COORD_T is <>;
with function "*"(A, B : COORD_T) return COORD_T is <>;
with function "/"(A, B : COORD_T) return COORD_T is <>;

--g-----------------------------------------------------------------------------------------g--
package RECTANGLE is
--g-----------------------------------------------------------------------------------------g--

  --!-----------------------------------------------------------------------------------------!--

  -- the coordinate frame is that of:        and RECTs are sorted/stored as:

  --                   y
  --                   |                                  .-----.CORNER_B
  --                   |                                  |     |
  --                   ._____ x                 CORNER_A._____.

  -- (note, X WINDOWS, for some reason has y running downwards, so some transforming should take place,
  -- when using X WINDOWS as the target drawing environment)

  -- this exception is raised when DEL fails, or if SIZE of an empty object is requested
  RECTANGLE_CONSTRAINT_ERROR : exception;


  -- some standard graphics objects
  type POINT_T is record
    X, Y : COORD_T;
  end record;

  type RECT_T is record
    CORNER_A, CORNER_B : POINT_T;
  end record;

  type RECTS_T is private;
```

```
--!-------------------------------------------------------------------------------------!--

function INIT(
  BOUNDS : COORD_ARR_T := DEFAULT_BOUNDS;
  THRESHOLD : LONG_INTEGER := ADD_QUAD_TREE_THRESHOLD_DIVIDE_FIGURE
) return RECTS_T;

procedure DESTROY(THIS : in out RECTS_T);

function VALID(RECTS : RECTS_T) return BOOLEAN;

-- for proper efficient (relatively speaking) destruction ..
generic
  with procedure DESTROY(THIS : in out DATA_T);
procedure DESTROY_TEMPLATE(THIS : in out RECTS_T);

procedure ADD(THIS_DATA : DATA_T; TO_THIS : RECTS_T);
procedure DEL(THIS_DATA : DATA_T; FROM_THIS : RECTS_T);

generic
  with procedure DESTROY(THIS : in out DATA_T);
procedure DEL_TEMPLATE(THIS_DATA : DATA_T; FROM_THIS : RECTS_T);

-- the extents of a RECTS_T are returned - useful for centering purposes
function EXTENT(THIS : RECTS_T) return RECT_T;
-- in a layered setup, the full extents will be the union of extents for each layer
function UNION(R1, R2 : RECT_T) return RECT_T;

function COORD2RECT(COORD : COORD_ARR_T) return RECT_T;
function RECT2COORD(RECT : RECT_T) return COORD_ARR_T;

-- note, the two generic iteraters below require a RECT_T over which the search will happen

--!-------------------------------------------------------------------------------------!--

generic
  -- these fns should return TRUE, iff the iteration is to stop.
  -- INVERSE - changes the sense of the search to objects outside the given region
  -- the rectangle passed, must be sorted
```

```
    -- CUT - called when a rectangle intersects with the given region
    -- NOT_CUT - called when rectangle is wholly enclosed/outside the region

    with function CUT(DATA : DATA_T) return BOOLEAN;
    with function NOT_CUT(DATA : DATA_T) return BOOLEAN;
  procedure ITERATE(DOMAIN : RECTS_T; REGION : RECT_T; INVERSE : BOOLEAN := FALSE);

  --!------------------------------------------------------------------------------------------!--

  -- this generic is similar to above, but, if some items are 'too' small, they don't get passed back
  generic
    -- INVISIBLE - the actual rectangle which has objects inside it, but are within DELTA_X and DELTA_Y
    with function INVISIBLE(RECT: RECT_T) return BOOLEAN;

    with function CUT(DATA : DATA_T) return BOOLEAN;
    with function NOT_CUT(DATA : DATA_T) return BOOLEAN;
  procedure ITERATE_LIMIT(
    DOMAIN : RECTS_T;
    REGION : RECT_T;
    DELTA_X, DELTA_Y : COORD_T;   -- the min size of objects that are acceptable
    INVERSE : BOOLEAN := FALSE
  );

  --!------------------------------------------------------------------------------------------!--

  -- this is for test purposes
  generic
    with function IMAGE(COORD : COORD_T) return STRING;
    with function INFO(DATA : DATA_T) return STRING;
  procedure DUMP(RECTS : RECTS_T);

private

  type RECTS_REC_T;
  type RECTS_T is access RECTS_REC_T;

end RECTANGLE;
```

Figure A-5. Rectangle Manager Generic

```
generic

  type SET_VALUE_T is (<>); -- I can handle integers or enumeration entities as the set value

--g-----------------------------------------------------------------------------------------------------g--
package SPARSE_SET is
--g-----------------------------------------------------------------------------------------------------g--

  -- This package supports fast and efficient *big* set handling.
  -- Thus, sets of millions of elements will be handled without a blink
  -- (The package works well when there are large contiguous ranges of values,
  -- since it only holds the start and finish)

  type SET_T is private;

  --!---------------------------------------------------------------------------------------------------!--

  function INIT return SET_T;
  function VALID(SET : SET_T) return BOOLEAN;

  --!---------------------------------------------------------------------------------------------------!--

  -- these form the 'basic' set operations
  function UNION(A, B : SET_T) return SET_T;
  function INTERSECTION(A, B : SET_T) return SET_T;
  function SUBTRACT(A, B : SET_T) return SET_T;
  -- WORLD must fully contain set A (ie., this is not checked for)
  function INVERSE(A, WORLD : SET_T) return SET_T;

  function EXISTS(IN_THIS : SET_T; THIS : SET_VALUE_T) return BOOLEAN;

  --!---------------------------------------------------------------------------------------------------!--

  procedure ADD(TO_THIS : SET_T; START, FINISH : SET_VALUE_T);
  procedure ADD(TO_THIS : SET_T; VALUE : SET_VALUE_T);

  procedure DEL(FROM_THIS : SET_T; VALUE : SET_VALUE_T);
  procedure DEL(FROM_THIS : SET_T; START, FINISH : SET_VALUE_T);

  procedure DESTROY(THIS : in out SET_T);
```

```
  function COPY(THIS : SET_T) return SET_T;

  -- this count routine could cause NUMBERIC_ERROR if the number of items is greater than LONG_INTEGER'LAST
  -- (which could happen if the set spans best part of LONG_INTEGER'RANGE rebased to start from 1)
  function SIZE(THIS : SET_T) return LONG_INTEGER;


  --!----------------------------------------------------------------------------------------------!--

  type DIRECTION_T is (LEFT_RIGHT, RIGHT_LEFT);

  -- the following two generics, when stamped, contine while REGISTER returns 'TRUE'
  generic
    with function REGISTER(VALUE : SET_VALUE_T) return BOOLEAN;
  procedure ITERATE(SET : SET_T; DIRECTION : DIRECTION_T := LEFT_RIGHT);


  -- this is provided as a concession to the internal implementation of the list
  -- since it removes, on average, a large number of additional function calls,
  -- and can be used for say displaying the elements of a set in a compressed form
  generic
    with function REGISTER(START, FINISH : SET_VALUE_T) return BOOLEAN;
  procedure COMPRESSED_ITERATE(SET : SET_T; DIRECTION : DIRECTION_T := LEFT_RIGHT);

private

  type SET_REC_T;
  type SET_T is access SET_REC_T;

end SPARSE_SET;
```

Figure A-6. Sets Generic

```
--*-----------------------------------------------------------------------------------------------------------*--
function UNION(A, B : SET_T) return SET_T is
--*-----------------------------------------------------------------------------------------------------------*--
  -- some types to aid in readability of which set is at the 'pivot' point
  type PIVOT_T is (A_set, B_set);
  PIVOT : PIVOT_T;
  NEW_SET : SET_T := INIT;
  A_POS, B_POS : LONG_INTEGER := 1; -- these are elements we intend to read, not 'have' read
  A_END : LONG_INTEGER := SIZE(A.SLICES);  B_END : LONG_INTEGER := SIZE(B.SLICES);
  A_LOOKAHEAD, B_LOOKAHEAD : SLICE_T;  -- used during the merging
  THE_MERGE : SLICE_T; -- this contains the 'final' slice to be added to NEW_SET
--[-----------------------------------------------------------------------------------------------------------[--
begin
  -- we know that both lists are ordered, so we start one side and flip between both
  loop
    -- these are our exit conditions
    exit when  A_POS > A_END and B_POS > B_END;

    -- if either one of the lists is 'ended', then we have a simple tack on job
    if A_POS > A_END then -- add B onto NEW_SET
      for i in B_POS..B_END loop
        ADD(NEW_SET.SLICES, THIS => GET(B.SLICES,i));
      end loop;
      -- now move on the value of B_POS, so the outer loop will terminate
      B_POS := B_END + 1;
    elsif B_POS > B_END then -- add A onto NEW_SET
      for i in A_POS..A_END loop
        ADD(NEW_SET.SLICES, THIS => GET(A.SLICES,i));
      end loop;
      -- now move on the value of B_POS, so the outer loop will terminate
      A_POS := A_END + 1;
    else -- something must be present in A and B sets
      -- first set up what we are trying to 'union' togethor
      A_LOOKAHEAD := GET(A.SLICES,A_POS);
      B_LOOKAHEAD := GET(B.SLICES,B_POS);
      -- now we decide which one will 'dominate'
      if A_LOOKAHEAD.START < B_LOOKAHEAD.START then -- A comes first, so merge with B set
        PIVOT := B_set; A_POS := A_POS + 1; THE_MERGE := A_LOOKAHEAD;
      else -- catches the 'equal' case -- B comes first, so merge with A set
        PIVOT := A_set; B_POS := B_POS + 1; THE_MERGE := B_LOOKAHEAD;
      end if;
```

```
-- when we get to this point, one of the LOOKAHEADs is out of date, thus
-- the following code has to watch out that on the 'flip' that there is info
-- ready to merge, otherwise, we call it a day since we're at the end of a SET

-- now we run along both sides merging what we can. on completion, we add THE_MERGE to NEW_SET.
-- there are three cases of merging possible, that need to be considered:
--
-- (1) |-----|      fully enclosed, which means carry on merging with second row
--        |==|
--
-- (2) |-----|      partially overlapped, this means flip the merge to elements on row1
--          |==|
--
--     |-----|      this case is also valid, eg. [1-4],[5-6] is the same as [1-6]
--             |==|
--
-- (3) |-----|      forget it, neither row1 or row2 has anything going
--              |==|

loop
  case PIVOT is -- the set that we are taking 'elements' out of to try and merge
    when A_set =>
      -- the following if statement is complicated, 'cause we don't want to hit the
      -- SET_VALUE_T'LAST limit when checking out the second case shown in (2) above
      if A_LOOKAHEAD.START <= THE_MERGE.FINISH
         or else A_LOOKAHEAD.START = SET_VALUE_T'SUCC(THE_MERGE.FINISH) then
      -- yes, it's a merge - but a 'full' one ?
        A_POS := A_POS + 1; -- this is saying, we 'accept' this element
        if A_LOOKAHEAD.FINISH > THE_MERGE.FINISH then -- is an overlap merge     ... (2)
          -- this dominates, so we need to *flip* PIVOT
          THE_MERGE.FINISH := A_LOOKAHEAD.FINISH;
          -- check if there are B elements to merge, if so, setup LOOKAHEAD
          if B_POS > B_END then exit; end if;
          B_LOOKAHEAD := GET(B.SLICES,B_POS);
          PIVOT := B_set; -- indicate the flip
        else -- A_LOOKHEAD is fully enclosed - no need to flip PIVOT,
             -- so lookahead on B again for more merges                          ... (1)
          -- check if there are A elements to merge, if so, setup LOOKAHEAD
          if A_POS > A_END then exit; end if;
          A_LOOKAHEAD := GET(A.SLICES,A_POS);
        end if;
```

```
          else -- that's it, there is 'too' big a gap to the next A element, so exit ... (3)
            exit;
          end if;
        when B_set =>
          -- the following if statement is complicated, 'cause we don't want to hit the
          -- SET_VALUE_T'LAST limit when checking out the second case shown in (2) above
          if B_LOOKAHEAD.START <= THE_MERGE.FINISH
             or else B_LOOKAHEAD.START = SET_VALUE_T'SUCC(THE_MERGE.FINISH) then
          -- yes, it's a merge - but a 'full' one ?
            B_POS := B_POS + 1; -- this is saying, we 'accept' this element
            if B_LOOKAHEAD.FINISH > THE_MERGE.FINISH then -- is an overlap merge     ... (2)
              -- this dominates, so we need to *flip* PIVOT
              THE_MERGE.FINISH := B_LOOKAHEAD.FINISH;
              -- check if there are A elements to merge, if so, setup LOOKAHEAD
              if A_POS > A_END then exit; end if;
              A_LOOKAHEAD := GET(A.SLICES,A_POS);
              PIVOT := A_set; -- indicate the flip
            else -- B_LOOKHEAD is fully enclosed - no need to flip PIVOT,
                 -- so lookahead on B again for more merges                          ... (1)
              -- check if there are B elements to merge, if so, setup LOOKAHEAD
              if B_POS > B_END then exit; end if;
              B_LOOKAHEAD := GET(B.SLICES,B_POS);
            end if;
          else -- that's it, there is 'too' big a gap to the next B element, so exit ... (3)
            exit;
          end if;
      end case;
    end loop;
    -- once we get here, we've 'sucked' in as much as possible, so we now add the element to NEW_SET
    ADD(NEW_SET.SLICES, THIS => THE_MERGE);
    -- now we carry on round the loop for the next load that we can try and merge
  end if;
end loop;
-- once we get here, the UNION should be completed, so ...
return NEW_SET;
end UNION;
--]------------------------------------------------------------------------------------]--
```

Figure A-7. Union Algorithm