



City Research Online

City St George's, University of London

Citation: Roberts, M. (1990). Visual programming for transputer systems.
(Unpublished Doctoral thesis, City, University of London)

This is the accepted version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: <https://openaccess.city.ac.uk/id/eprint/28547/>

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to [City Research Online policy](#).

Visual programming for Transputer systems

by

Mike Roberts

Thesis submitted for the degree of
Doctor of Philosophy

THE CITY UNIVERSITY
Centre for Information Engineering

November, 1990

Table of Contents

Chapter 1 - Graphical tools for parallel programming

1.0 Introduction	1-1
1.1 Objectives and synopsis	1-3
1.2 Thesis outline	1-4

Chapter 2 - A review of related work

2.0 Introduction	2-1
2.1 Methodological approaches to parallelism	2-1
2.2 Work carried out on visual program development tools for parallel systems	2-2
2.3 Theoretical aspects of visual tools	2-2
2.4 Visual program development technologies	2-3
2.4.1 Visual programming systems for parallel computation	2-3
2.4.1.1 Petri-net based systems	2-4
2.4.1.2 Dataflow and functionally based visual programming systems	2-5
2.4.1.3 Coarse grain dataflow based visual programming systems	2-7
2.4.1.4 CSP and CCS based visual programming systems	2-8
2.4.1.5 Other visual programming systems	2-9
2.4.2 Graphical CASE tools and visual simulation systems	2-13
2.4.3 Program visualisation systems for parallel machines	2-14
2.4.3.1 Static program visualisation systems	2-15
2.4.3.2 Dynamic program visualisation systems	2-17

2.4.3.2.1 Static program visualisation with dynamic overlay of information	2-17
2.4.3.2.2 Truly dynamic program visualisation systems	2-18
2.5 Visual process to processor mapping tools	2-20
2.6 Review summary	2-20

Chapter 3 - Visualisations of Occam programs and Transputer systems

3.0 Introduction	3-1
3.1 The Occam programming language	3-1
3.1.1 Data types, variable definitions and constants	3-2
3.1.2 Channel protocols	3-2
3.1.3 Channels	3-2
3.1.4 Primitive processes	3-3
3.1.4.1 Assignment	3-3
3.1.4.2 Output	3-3
3.1.4.3 Input	3-3
3.1.5 Scope of channels and variables	3-4
3.1.6 SKIP and STOP processes	3-4
3.1.7 Constructed processes	3-4
3.1.7.1 Sequential construct (sequence)	3-4
3.1.7.2 Conditional processes	3-5
3.1.7.3 Parallel construct	3-5
3.1.7.4 Alternative processes	3-6
3.1.8 Comments	3-7
3.1.9 Procedural abstraction	3-7
3.1.10 Loops and replicators	3-7
3.1.11 An example Occam program	3-8
3.2 The Transputer	3-9
3.3 Configuration of Occam programs for execution on Transputer systems	3-10
3.4 Visualisations of Occam and of transputer arrays	3-12
3.4.1 Rack diagrams	3-13

3.4.2 Bubble and arc diagrams	3-14
3.4.3 Control flow diagrams	3-16
3.5 Features of a visualisation of Occam	3-18
3.5.1 Visual representation of processes	3-18
3.5.2 Mixed representation of inter-process communication and control flow	3-18
3.5.3 Use of graphics for high level overviews and text for low level detail	3-19
3.5.4 Visual hierarchy	3-19
3.6 The GILT language	3-19
3.6.1 General design principles	3-19
3.6.2 An overview of the GILT system	3-21
3.6.3 GILT diagrams	3-23
 Chapter 4 - Formal descriptions of visual languages	
4.0 Introduction	4-1
4.1 Language, communication, and computer languages	4-1
4.2 Specification of computer languages	4-2
4.3 Grammars and syntactic specification	4-2
4.3.1 Text grammars and the specification of textual language syntax	4-2
4.3.2 Graph grammars for the specification of visual language syntax	4-4
4.3.2.1 Previous work on the syntactic descriptions of visual languages	4-4
4.3.2.2 Graph grammars	4-6
4.3.2.3 A simple visual language grammar	4-8
4.3.2.4 A graph grammar for a subset of GILT	4-10
4.3.2.5 Hierarchy and communication	4-11
4.3.2.5.1 Hierarchy	4-14
4.3.2.5.1.1 H-graphs	4-15
4.3.2.5.1.2 An H-graph grammar for a subset of GILT	4-16

4.3.2.5.1.3 The simple grammar with hierarchy	4-16
4.3.2.5.2 Communication	4-16
4.3.2.6 Omissions of the grammar	4-24
4.3.2.7 GILT graph symbols and their relationship to nodes and arcs	4-28
4.3.3 Describing the textual parts of the GILT language	4-28
4.4 Semantics of visual languages and the semantic definition of GILT	4-29

Chapter 5

5.0 Introduction	5-1
5.1 The GILT language	5-1
5.2 GILT diagrams	5-2
5.2.1 Functional icon components	5-3
5.2.1.1 Ports	5-3
5.2.1.1.1 Control Flow Ports	5-3
5.2.1.1.2 Not Control Flow Output Ports	5-4
5.2.1.1.3 Channel Input and Output Ports	5-5
5.2.1.2 Text Areas	5-5
5.2.1.2.1 Name Text Areas	5-5
5.2.1.2.2 Expression Text Areas	5-6
5.2.1.2.3 Condition Text Areas	5-7
5.2.1.2.4 Shutoff Text Areas	5-7
5.2.1.2.5 Input Variable Text Areas	5-8
5.2.1.2.6 Variable Declaration Text Areas	5-8
5.2.2 Links	5-9
5.2.3 Functional Icons	5-10
5.2.3.1 Functional Icons which do not form part of constructs	5-10
5.2.3.1.1 Comment	5-10
5.2.3.1.2 Variable Declaration Icons	5-20

5.2.3.2 Functional Icons which form parts of constructs	5-20
5.2.3.2.1 Process Icon instances	5-20
5.2.3.2.2 Stubs	5-21
5.2.3.2.2.1 Control Input and Control Output Stubs	5-21
5.2.3.2.2.2 Channel Input and Output Stubs	5-21
5.2.3.2.3 Condition Icons	5-22
5.2.3.2.4 Guard Icons	5-22
5.2.3.2.5 Declared Parameter Icons	5-23
5.2.3.2.6 Passed Parameter Icons	5-23
5.2.3.2.7 Channel Connector Icons	5-23
5.2.3.2.8 Control Split Join Icons	5-23
5.2.4 Visual abstraction and the definition of Process Icons	5-24
5.2.5 Constructs	5-29
5.2.5.1 Control flow constructs	5-29
5.2.5.1.1 The stop and skip constructs	5-30
5.2.5.1.2 Unconstructed process	5-30
5.2.5.1.3 Sequential construct (sequence)	5-31
5.2.5.1.4 Parallel construct	5-32
5.2.5.1.5 Alternative construct	5-34
5.2.5.1.6 Conditional construct	5-34
5.2.5.1.7 While construct	5-35
5.2.5.1.8 Other control flow constructs	5-35
5.2.5.2 Inter-process communication constructs	5-36
5.3 Textual process specifications	5-41
5.4 Programming with GILT	5-45
5.4.1 A processor farm in GILT	5-46
5.4.2 A circular buffer in GILT	5-53

Chapter 6 - An editor for GILT diagrams

6.0 Introduction	6-1
6.1 Editing system overview	6-1
6.2 Systems for the implementation of the editor	6-2
6.3 The GILT program editing system	6-3
6.3.1 The graphics editor	6-3
6.3.1.1 The Process Icon editor	6-6
6.3.1.1.1 Process Icon editor controls	6-7
6.3.1.2 The Process Icon library (browser)	6-9
6.3.1.2.1 Process Icon library controls	6-9
6.3.1.3 The diagram editor	6-10
6.3.1.3.1 Diagram editor controls	6-10
6.3.1.3.1.1 Mode controls and editing functions	6-12
6.3.1.3.1.1.1 Adding diagram components to a definition diagram	6-14
6.3.1.3.1.1.2 Deleting diagram components	6-14
6.3.1.3.1.1.3 Editing the textual or diagrammatic definition of a Process Icon	6-14
6.3.1.3.1.1.4 Editing the contents of text areas	6-15
6.3.1.3.1.2 Toggle controls	6-15
6.3.1.3.1.3 System controls	6-16
6.3.2 The text editing system	6-16
6.3.2.1 Functional description of the text editor	6-16
6.3.3 Implementation of the editor	6-19
6.3.3.1 Relevant features of the sunview system	6-19
6.3.3.2 Implementation of the Process Icon editor	6-22
6.3.3.3 Implementation of the Process Icon library	6-23
6.3.3.4 Implementation of the diagram editor	6-25
6.3.3.5 Implementation of the text editor	6-29

Chapter 7 - A compiler for the GILT language

7.0 Introduction	7-1
7.1 Previous work on compilation systems for visual languages	7-1
7.2 Compiler overview	7-1
7.3 Choice of implementation language	7-3
7.4 A very brief introduction to Prolog	7-4
7.5 The tokeniser	7-5
7.5.1 Definition identifiers and associated tokens	7-6
7.5.2 Diagram component identifiers and associated tokens	7-7
7.5.3 Hierarchy tokens	7-11
7.5.4 Example output from the tokeniser	7-13
7.6 Transfer of tokens between the tokeniser and the parser	7-13
7.7 Parsing	7-17
7.7.1 Graph reductions and the parsing of GILT diagrams	7-17
7.7.2 Parsing communications structures, non-channel parameter structures and local variable definitions	7-19
7.7.3 Parsing the control flow graphs	7-21
7.7.4 Error checking	7-32
7.8 Code Generation	7-32
7.9 Interaction with the compiler	7-36
7.9.1 The control panel	7-38
7.9.2 The compiler subwindow	7-39

Chapter 8 - Results, conclusions and suggestions

8.0 Introduction	8-1
8.1 Results and conclusions	8-1
8.1.1 A visual programming system for parallel computation	8-1

8.1.2 The use of graph grammars for visual language syntax	8-2
8.1.3 Construction of visual programming systems from standard user interface components	8-3
8.1.4 A compiler for a visual language	8-3
8.1.5 Summary of conclusions and results	8-3
8.2 Suggestions for future work	8-4
8.2.1 Improvements and enhancements to the existing system	8-4
8.2.1.1 Checking of textual syntactic entities	8-4
8.2.1.2 Configuration of GILT programs	8-5
8.2.1.3 More advanced editing facilities	8-5
8.2.1.4 Support for channel protocols	8-5
8.2.1.5 Parsing GILT diagrams	8-6
8.2.1.6 Support for the use of replicators	8-7
8.2.1.7 Animated execution of GILT programs	8-7
8.2.2 Working towards future graphical program development tools	8-7
8.2.2.1 Mixed paradigm approaches	8-8
8.2.2.2 Combined visual programming and program visualisation	8-8
8.2.2.3 Visual mixed language programming	8-8
8.2.3.4 The use of colour	8-8
8.2.3.5 Multidimensional tools	8-8

Appendix 1

Appendix 2

Appendix 3

References

List of Figures

Chapter 1

- 1.1 A block diagram of the GILT system. 1-4

Chapter 2

- 2.1 An example function from the PROGRAPH visual programming system. 2-6
- 2.2 An example inter-process communication diagram from the IPIGS system which shows a single process communicating with an array of five processes. 2-9
- 2.3 Clara pictorial CCS expression showing four processes ("PROC1", "PROC2", "SEM" and "REG") whose behavior is defined by four pictorial expressions. 2-10
- 2.4 An example application program in HI-VISUAL. 2-11
- 2.5 A screen from GRAIL showing an Occam process and associated channel connections. 2-16

Chapter 3

- 3.1 An example Occam program, demonstrating the use of buffering process "buffer.1" in a queuing system. 3-8
- 3.2 A configured version of the program in figure 3.1. 3-11
- 3.3 The network topology defined by the configured Occam program of figure 3.2. 3-12
- 3.4 A Rack Diagram showing sixteen transputers wired into double ring topology. 3-13
- 3.5 A "bubble and arc" diagram from Inmos (1989). 3-15
- 3.6 A control flow diagram with overlaid inter-process communication information from Mourlin and Cournarie (1989). 3-17
- 3.7 A GILT Process Icon representing a data display process. 3-24

3.8	A GILT definition diagram defining the functionality of the Process Icon of figure 3.7.	3-25
-----	---	------

Chapter 4

4.1	Restricted GILT grammar.	4-12
4.2	An example graph generated using the grammar of figure 4.1.	4-14
4.3	Hierarchical restricted GILT grammar.	4-17
4.4	Example hierarchical graph.	4-19
4.5	A generalised $n \times n$ processor array, such as might be used in a pattern matching algorithm.	4-21
4.6	Productions describing communications graphs.	4-22
4.7	Example communication graphs.	4-23
4.8	Productions describing process connectivity with channel input and output ports.	4-25
4.9	Example base graph.	4-26
4.10	Total graph.	4-27

Chapter 5

5.1	Ports in the current implementation of GILT together with their representative terminal symbols in the grammar.	5-4
5.2	Examples of Text Areas and their representative terminal symbols in the grammar.	5-6
5.3	Line styles used in GILT diagrams and their representation in the syntax.	5-9
5.4	The equivalence between functional icons and their grammatical representations.	5-11
5.5	The productions in the base grammar describing the GILT language.	5-13
5.6	A definition diagram with two parallel constructs in sequence using the original representation.	5-24
5.7	A definition diagram with two parallel constructs in sequence using the current representation.	5-25

5.8	A Process Icon instance and its definition diagram, showing the correspondence between stubs and ports.	5-28
5.9	An illustration of GILT's non-channel parameter passing mechanism.	5-29
5.10	The stop construct in GILT.	5-31
5.11	GILT's skip construct.	5-31
5.12	An example sequential construct.	5-32
5.13	An example parallel construct.	5-33
5.14	An example alternative construct.	5-33
5.15	A pair of example conditional constructs	5-35
5.16	An example while construct.	5-36
5.17	A communications graph such as might have been produced in early versions of the GILT system.	5-38
5.18	The communication graph of figure 5.17 with an added Channel Output Port and Channel Link.	5-39
5.19	A completely connected version of figure 5.18.	5-40
5.20	Productions in a communications grammar which may be used to generate the graph of figure 5.21.	5-42
5.21	An equivalent graph to that of figure 5.10 produced with the grammar of figure 5.20.	5-43
5.22	A GILT diagram having a communications structure like the one in figure 5.21.	5-43
5.23	An example text window showing a single element buffer process with one Channel Input Stub and one Channel Output Stub.	5-44
5.24	Occam for the main part of a processor farm.	5-47
5.25	Two "top-level" views of a processor farm in GILT.	5-48
5.26	A view of the farm without specialised icons.	5-49
5.27	Definition diagram for "farmer" Process Icon.	5-49
5.28	Definition diagram for "worker" Process Icon.	5-50
5.29	Definition diagram for "end.worker" Process Icon.	5-51
5.30	Occam for a "farmer" process.	5-51
5.31	Occam for a "worker" process.	5-52

5.32	Occam for an "end.worker" process.	5-52
5.33	Occam code implementing the circular buffer algorithm.	5-54
5.34	A top-level view of the circular buffer example.	5-56
5.35	A text editing window for the Process Icon "buffer.1".	5-57
5.36	The definition diagram for the Process Icon "most".	5-58
5.37	The definition diagram for the Process Icon "do.circle".	5-58
5.38	The definition diagram for the Process Icon "putnget".	5-59
5.39	A text window for the Process Icon "put".	5-60
5.40	A text window for the Process Icon "get".	5-61

Chapter 6

6.1	A screendump showing the main window from the diagram and icon editing system.	6-4
6.2	The pop-up Process Icon library showing three newly created (and unedited) Process Icons ready for selection.	6-5
6.3	A pop-up window requesting confirmation of a delete action.	6-5
6.4	A pop-up window giving assistance on the use of the system.	6-5
6.5	A close up view of the Process Icon editor control panel.	6-7
6.6	A close up view of the controls for the Process Icon library.	6-10
6.7	An enlarged view of the diagram editor parts of the editor display.	6-11
6.8	A text window.	6-17
6.9	The flow of control in a typical Sunview applications program.	6-20
6.10	How items in the browser reference Process Icon definitions stored in definition node data structures.	6-24
6.11	The pointers between definition nodes and instance nodes.	6-24
6.12	The representation of Channel Links in the GILT program editors' internal data structures.	6-26
6.13	Separate item descriptions referencing the same instance node.	6-27

Chapter 7

7.1	A block diagram of the compiler for the GILT language.	7-1
7.2	The relationship between diagram component nodes and definition nodes.	7-12
7.3	A view of the instance-definition tokens, showing how diagram component identifiers for Process Icon instances are related to the definition identifiers for their Process Icon definition.	7-13
7.4	Output from the tokeniser for the double buffer example, the definition diagram of which is shown in figure 7.5.	7-14
7.5	A definition diagram for the double buffer example, labelled with numeric labels for reference to figure 7.4.	7-16
7.6	Prolog facts containing lists generated by the parser for the definition node with definition identifier "450656" of figure 7.4.	7-20
7.7	Prolog facts containing lists generated for the two Process Icon instances in the definition diagram of figure 7.5.	7-21
7.8	The state of the Prolog database on entry to the part of the parser concerned with parsing the control flow structures.	7-22
7.9	The reduction of a control flow graph showing how a parse tree for the graph is constructed.	7-24
7.10	A parse tree for the control flow graph (a) of figure 7.9.	7-25
7.11	Equivalent parallel constructs in GILT.	7-26
7.12	Two equivalent parse trees.	7-27
7.13	A parse tree for an n-way parallel construct, making use of construct associativity.	7-27
7.14	A parse of the control flow graph of figure 7.19(a) using the associativity of constructs to simplify the reductions used in parsing.	7-29
7.15	An example graph reduction routine, encoded in Prolog, which recognises parts of parallel constructs.	7-31
7.16	State of the Prolog internal database on entry to the code generation routine.	7-33
7.17	Output from the compiler for the two buffer example.	7-37

7.18	Output from the complier for the two buffer example modified so that the code produced for non-procedural processes may be examined.	7-37
7.19	The complete compiler window is shown in its initial state.	7-38
7.20	The complier window showing an error message from the complier indicating a node which has not been reduced, and thus an erroneous diagram.	7-40
7.21	A view of an erroneous diagram showing a functional icon highlighted by the use of the error location facility.	7-40
7.22	Successful compilation message giving the filename containing the compiler output.	7-41
7.23	An example warning message.	7-41

List of Tables

Chapter 7

- | | | |
|-----|--|------|
| 7.1 | The equivalence between the valued nodes of GILT's syntax and the Prolog facts or tokens which are output to represent them. | 7-9. |
|-----|--|------|

Acknowledgements

I would like to thank the following people for the help and support over the past years:

My supervisor, Pat Samwell, for her patience, her encouragement and for providing me with the equipment on which I performed my work.

Tim Ellis, Paul Rosin, Geoff West and Beatrice Brillault of the machine vision group at City University for their peculiar humour, Prolog, mathematical discussions and several thousand sheets of printer paper.

Dave Styles, for providing encouragement from "next door" at my darkest moments.

Alan Godard for ventura style sheets and other tips.

My very good friends Tim Smith, for a seemingly endless supply of surfboards, and Johnny Farrington, for his strangeness and the "bee" icons.

Especially my then girlfriend now wife, Melissa Miller, for laughter, love, forgetting and for not throwing my thesis at me!

Finally, my father for his continued financial support and for being everything a good father should be.

The author acknowledges the support of the SERC (U.K. Science and Education Research Council).

Declaration

I grant powers of discretion to the University Librarian to allow this thesis to be copied in whole or in part without further reference to me. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Abstract

The techniques of visual programming, in which programs are constructed using graphical representations, have much to offer concurrency. This thesis reports on work performed during the development of the visual programming language, GILT (GraphIcal Language for Transputers). GILT uses a mixed text-graphics paradigm to aid the parallel programming process. It is strongly hierarchical and mixes visualisations of Occam style processes, inter-process communication and control flow to yield new representations of concurrent programming structures. GILT's syntax is fully defined using graph grammars and extended BNF, which together provide a new syntactic formalism for visual languages which have a mixed text-graphics model.

To support the production of GILT programs, a prototype environment has been developed. The environment, which has been developed on a Sun workstation, consists of a program editor and a compiler within an integrated runtime environment. The editor has been constructed using standard user interface components and it is shown that such components are well suited to the rapid prototyping of visual languages. GILT's compiler uses a graph reduction principle which is applicable to other visual languages and produces Occam as its output.

Parallel programming is a significantly complex matter for which definitive solutions will not be produced in the near future. This thesis therefore concentrates on the development of a unified set of techniques for the production of visual languages which are aimed at easing the problems of parallel programming.

Graphical tools for parallel programming

1.0 Introduction

Parallel (concurrent) programming is becoming increasingly important as multiprocessor based computers rapidly become viable alternatives to more traditional, uniprocessor, machines. Technological limitations on the computational power which may be attained by a single processor signify the eventual replacement of uniprocessor machines with parallel computers in very many areas.

Writing programs for parallel computers is an extremely complicated and problematical matter. Significant problems are caused by the inherent complexity of parallel systems and the interactions between their potentially myriad components, though even the behaviour of very simple parallel systems can be astonishingly complex. Diverse works have noted that parallelism is a particularly difficult area for programmers to work in. Programmers used to conventional sequential programming often find the conversion to "thinking parallel" quite difficult and tedious. In part this may be due to their conditioning into "thinking sequential" while programming and a reflection of the effect that the classical single processor von-Neumann architecture has had upon computing (Iannucci, 1983).

Although parallelism is not a new concept, the emergence of formally well founded methodologies and tools to aid the design and analysis of concurrent systems by overcoming the problems mentioned above has been relatively recent. The development of further tools and methodologies is essential to manage the complexity of parallel processing and aid the wide usage of parallel machines.

Methodologies like Hoare's Communicating Sequential Processes, CSP, (Hoare, 1985) and Milner's Calculus of Communicating Systems, CCS, (Milner, 1980) have advanced the study of parallelism and laid a theoretical framework for the design and study of the next generation of parallel systems.

Many tools have been developed for sequential programming, but the need for new parallelism specific tools aiding the concurrent programming process has been extensively noted, for example in (Cavano, 1988). Visual (graphical) program

development tools, which use computer graphics to aid the software development process, are one class of tools which have received increasing attention lately. The development of such tools is often motivated by arguments drawing on hopes of better exploiting the capabilities of the right hand side of the human brain, which is currently under utilised in programming (Shu, 1988). To understand why this is so it is necessary to examine the distribution within the brain of the highly specialised cerebral functions. These functions are in general asymmetrically divided between the right and left hemispheres of the brain, which function independently of each other to a certain degree. The left cerebral hemisphere is commonly thought of as a sequential information processor with specialised areas for verbal (spoken and written) expression. It is this side which is traditionally used in textually based programming. The right side of human brain is held to think in a more intuitive and artistic sense, for example perceiving melodies and other complex non-verbal patterns. Images captured in toto are analysed in such and the right side of the brain is seemingly capable of parallel processing. Perhaps the use of an area of the brain capable of "thinking in parallel" might be useful for the programming of concurrent computers? It has also been observed (Backus, 1977) that concurrency introduces an "extra dimension" to programming not present in conventional sequential programming, making the investigation of multi-dimensional (visual) tools for the development of parallel programs seem natural.

Some authors have gone so far as to suggest that graphics can completely replace text in the programming process. It must be remembered that graphics cannot supplant text in very many situations. Badly presented graphical representations are at best confusing and at worst unintelligible, while textual representations are essential for the expression of fully abstract concepts. Graphical representations and textual representations should be seen as complementing each other perfectly. Annotated diagrams are a primary example of this fact, which is often forgotten in anonymous quotations like "a picture is worth a thousand words".

Visual program development tools for parallel systems may be divided into three categories, visual process to processor mapping tools, visual programming tools and program visualisation tools. The definitions of the terms "visual programming" and "program visualisation" are due to Myers (1988).

Visual process to processor mapping tools are specific to parallel systems. They encompass systems which allows the users to specify the mapping of processes to processors in a two (or more) dimensional fashion. Visual process to processor mapping tools are closely related to visual programming tools, and in some senses can be considered a subset of visual programming tools. They are considered separately in this thesis.

Visual programming refers to any system that allows the user to specify a program in a two (or more) dimensional fashion. Conventional, textual, languages are not considered two dimensional since their compilers and interpreters process them as long one dimensional streams. Visual programming includes graphical programming languages and the use of conventional flow charts to create

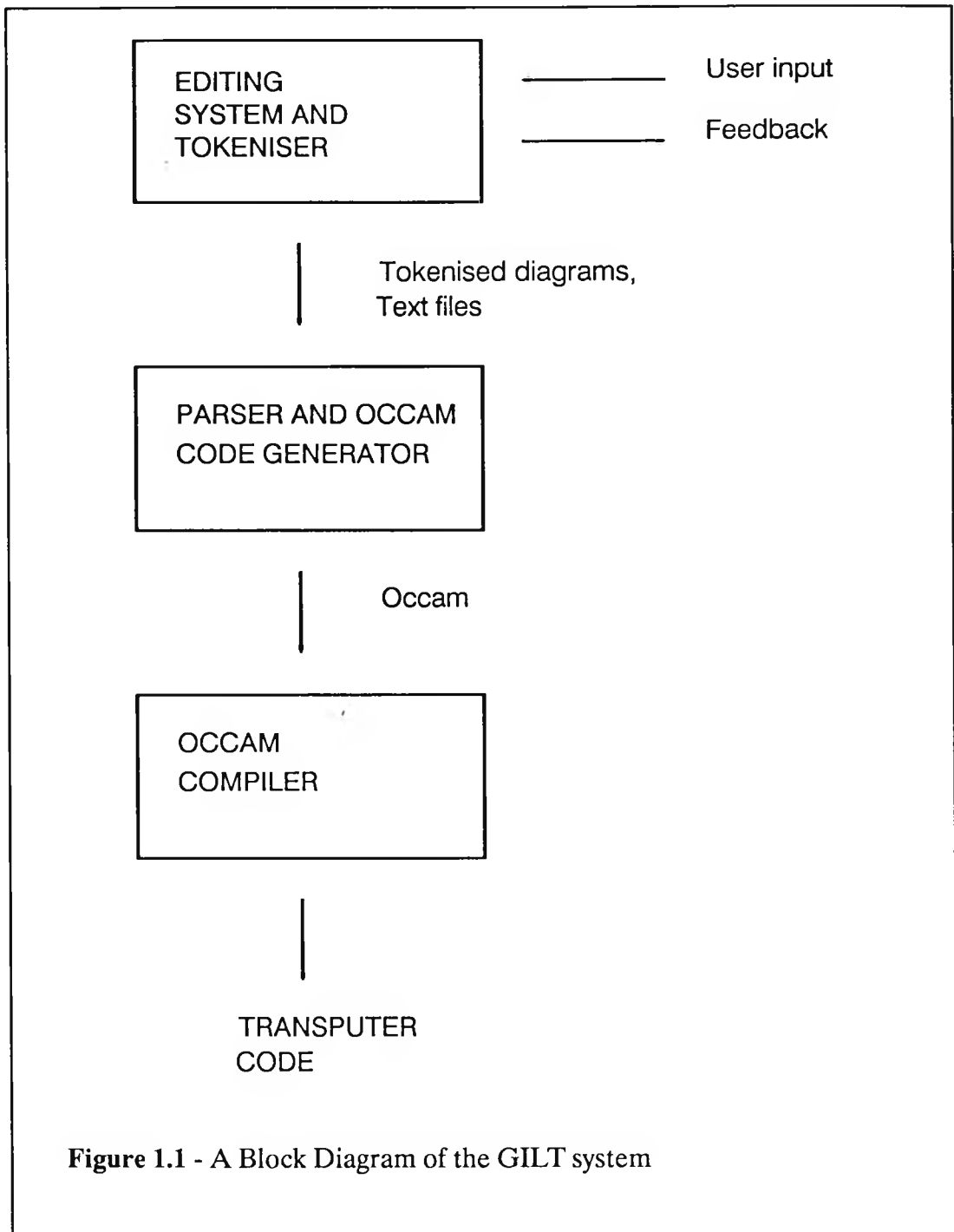
programs. It does not include systems that use conventional (linear) programming languages to define pictures.

Program visualisation is an entirely different concept from visual programming. In visual programming, the graphics is the program itself, but in program visualisation the program is specified in a conventional textual manner, with graphics used to illustrate some aspect of the program or its runtime execution. If a program created using visual programming is to be displayed or debugged, clearly this would be done in a graphical manner, but this would not be considered to be program visualisation.

1.1 Objectives and synopsis

The aim of the work reported in this thesis was to investigate how the techniques of visual programming could be applied to the development of parallel software for Transputer (Inmos, 1989a) systems. Although the investigation was to be Transputer and Occam (May, 1987) specific, as many generalised principles as possible were to be extracted for future use. To illustrate the techniques developed during the research a prototype "demonstrator" system was to be created. Any such system requires three basic components; a language specification, an editing system and a translation or compilation system. These components form the backbone of the work reported in the thesis.

The thesis describes the development of the demonstrator visual programming language "GILT" (GraphIcal Language for Transputers) together with its supporting software and the general techniques created for its implementation. GILT is a language relying on the close integration of text and graphics. It combines different visualisations of Occam programs to create a unique new system for the development of parallel software. GILT's approach allows visual programming to be used throughout the programming process except at the very lowest levels, which are performed using Occam. It also provides convenient visualisations for concurrent programming structures (such as the guarded execution of processes) not previously possible. To reflect GILT's mixed textual/graphical nature, its syntax relies upon context free graph grammars (for the visual parts of the language) and on context free text grammars (for the textual parts of the language). An integrated environment for GILT programming has been designed and implemented on a Sun 4/110 workstation (Sun, 1988). The environment makes heavy use of the Sun's graphics capabilities. The environment's program editing system is written in C, while the majority of the compiler (with the exception of the tokeniser) has been constructed using Prolog. A novel parsing method, which is based on graph reduction, and is suited to visual languages, is used in the compiler. The compiler produces Occam code suitable for further compilation using the Inmos Transputer Development System (Inmos, 1989), available on a specialised Transputer system (Transtech, 1989) attached to the Sun 4/110 used for development of GILT's environment. Software for the management of the user interface to the compilation system has also been written together with an error detection and display subsystem for the compiler. Figure 1.1 shows a block diagram for the software developed to support GILT.



1.2 Thesis outline

Chapter two is concerned with previous work relevant to the thesis. Methodologies for the design and implementation of concurrent systems are discussed together with their relationship to graphical program development tools. Examples of such tools for parallel systems, together with a few relevant sequential systems, are discussed and categorised.

Chapter three introduces Occam and the Transputer, discusses visualisations of Occam programs and Transputer arrays and analyses their features. A brief introduction to the GILT language is given together with the major decisions taken in its design. The relationship between the features of GILT and the analysis earlier in the chapter is clarified.

Chapter four examines approaches to the development of a formalised syntactic and semantic definition for visual languages based on a mixed textual/graphical paradigm similar to that used by GILT, drawing on the brief introduction of chapter three. It forms a basis for a full definition of the language and demonstrates how the visual constructs and symbols used in GILT may be modelled using graph grammars. The approach used for GILT's syntax is a new one and is based on the use of context free graph grammars and context free text grammars.

Chapter five gives a full definition of the GILT language. The syntax of the language is presented and discussed together with an informal semantic definition of the language's constructs, which is based on the correspondence between constructs in GILT and those in Occam. Example GILT programs are also given and discussed, showing how GILT can aid the development of Transputer applications.

Chapter six gives details on the editing system developed for the creation of GILT programs which is, unlike previous visual language editing systems, based on the use of conventional user interface components. The Sunview system, which was used for the implementation of the editing system, is described as are the data structures used for the internal representation of the diagrams.

Chapter seven deals with the compilation of GILT diagrams into Occam, describing the tokeniser, parser and code generator which have been developed. As a conventional approach to tokenisation and parsing is not possible for visual languages like GILT, new approaches are developed and described, based on the graph grammar syntax representation developed in chapter four.

Chapter eight summarises the results obtained from the work described in the previous chapters of the thesis, provides suggestions for future related work and draws conclusions.

In addition, appendices are included giving the syntax of Occam (appendix one), a formal definition of the graphs and graph grammars used in chapter four and five (appendix two), and published work relevant to the thesis (appendix three).

2

A review of related work

2.0 Introduction

This chapter briefly outlines approaches to the development of parallel software, their importance and their relationship to tools and to visual tools in particular. Specific graphical tools are then discussed and classified into various categories for analysis.

Some tools not specifically concerned with the development of parallel programs are also included, where they are considered to be useful and relevant.

2.1 Methodological approaches to parallelism

Mathematical models of parallelism are clearly desirable and are now seen as an essential basis for the design of reliable parallel systems. Hoare's CSP work (Hoare, 1985) is the basis of the Occam programming language (May, 1987) and is deeply concerned with the material of this thesis. In other fields of computer science, formal methods are finding widespread acceptance for system development. For example, the VDM (Jones, 1990) and Z (Spivey, 1989) methodologies are now commonly used in the formal definition of software systems. Further acceptance of such formal methods will mainly depend on two factors - the skills of the available work force and the ease of use of formally based systems and languages. The development of formally based tools to ease the use of rigorous methods in software design and implementation is vital.

Occam (May, 1987) is a language for concurrent programming which is based on CSP. Having such a well founded formalism underlying a language confers certain advantages. For example, it is possible to prove the correctness of small programs, and to apply the techniques of program transformation to the enhancement of performance (Roscoe and Hoare, 1986). Relationships between methodologies and languages like that between CSP and Occam are increasingly seen as desirable.

Other, higher level, tools using well founded design mechanisms seek to abstract design from implementation. Such computer aided software engineering (CASE) tools have been developed for distributed and parallel systems (Shatz and Wang, 1989). These tools seek to apply methodologies at a much higher level of abstraction than do languages based on formalisms.

Visual program development tools have the potential to confer considerable advantages on software development tools throughout the software development and maintenance process. They can offer more humanistic interfaces to formally well founded methods and thus facilitate the greater use of such methods. Several reasons for the adoption of visual program development tools were discussed in chapter one.

2.2 Work carried out on visual program development tools for parallel systems

Although the origins of the study of visual program development tools go back to the 1960's, the field is generally regarded as a recent sub-discipline of computer science because suitable technology allowing the widespread use of visual tools in the programming environment has only recently become available. Mid eighties systems like Pict (Glinert and Tanimoto, 1984) were extremely limited, aimed at novice programmers and concerned with sequential programming, but more recent developments have centred on aiding professional programmers.

Most recent review papers have been concerned with the wide areas to which graphical program development tools have been applied and have not in the main focused on visual tools for use with parallel systems. Of such works, developments in the visual programming and program visualisation fields are discussed in (Myers, 1988), (Shu, 1988) and in (Raeder, 1985). Few of the systems reviewed in the papers have been concerned with the generation and maintenance of parallel programs, instead focusing on sequential or object oriented programming systems. Ambler and Burnett (1989) discuss landmark systems and the impact of available technology on the development of visual tools.

2.3 Theoretical aspects of visual tools

Many early visual programming systems lacked a methodological approach. Research concentrated on the benefits conferred by visual representation. Systems without grammars or any form of language specification were also produced. It is, however, reassuring to see a growing, more formal, approach to the specification of visual languages. Graph grammars in particular have found application in the specification of the syntax of visual languages. A graph grammar may define the syntax of a visual language in the same way that a conventional context free grammar may define the syntax of a string language. Graph grammars and their relationships with visual languages and graphical tools are discussed later in the thesis, but several authors have produced important work worthy of mention here. Possibly the most notable, (Harel 1988), discusses work on Hi-Graph theory. Harel's work provides the basis of a formal method for describing the syntax of many diagrams. Hekmatpour and Woodman (1987) describe a syntax directed editing system for the development of graphical languages, aiming to cut down on the time required to implement a visual language editor. A similar approach is taken by Gottler (1989) in which an editing system based on context dependent graph grammars, whose productions may be visually input, is described. Harada

and Kunii (1984) discusses recursive graph theory as a formal basis for a visual design system.

Many visual programming systems rely heavily on the techniques of iconics (Lodding, 1983). Icons are commonly used for the representation of data, programs, processes, etc. Attempts have been made to produce formal models of iconic systems. Perhaps the best known work is (Chang, Tortora, Yu and Guercio, 1987). Chang et al separate generalised icons into a logical part (concerned with the meaning of the icon) and a physical part (the image). Iconic "operators" define an "icon algebra" for the construction of complex icons and specification of semantics for iconic constructs. A language generation system for purely iconic systems based on the work described above has also been produced (Chang, Tauber, Yu and Yu, 1987).

2.4 Visual program development technologies

In addition to graphical computer aided software engineering (CASE) tools two main types of graphical program development tool are acknowledged in literature on the subject - visual programming tools and program visualisation tools. Recently, however, a new class of tools unique to concurrent programming can be seen emerging. These tools deal with the problems of mapping processes to processors in a visual way. They will be referred to here as visual process to processor mapping tools and have much in common with the techniques of visual programming. Other systems concerned with the simulation of parallel machines and software also have some relevance.

2.4.1 Visual programming systems for parallel computation

The term "visual programming" refers to any system that allows the user to specify a program in a two or more dimensional fashion. Many visual programming systems have been produced for conventional sequential and object oriented systems. Surprisingly few systems have been developed for use in concurrent programming, and it is possible to review all major work in the area.

Although the visual programming systems discussed here differ from each other in many respects it is possible to identify a few common trends within the following systems.

Firstly, the majority of parallel visual programming languages use visual programming for the specification of static programming structures. These structures, for example fixed inter-processor communications pathways or dataflow networks, do not change with time. Such languages will be termed "static visual programming languages". The production of "dynamic visual programming languages", which are concerned with dynamically evolving structures, is a difficult task having more in common with the techniques of program visualisation, discussed later in this chapter. This is probably because the majority of textual concurrent programming languages, such as Occam, are concerned with the

specification of fixed inter-process communications structures. Only one notable system concerned with dynamic visual programming has emerged (Kaplan, Goering and Campbell 1989). In this system, dynamically evolving parallel structures are described through the use of a graphically displayed graph grammar, which constrains the connections that may evolve between processes, and textual process descriptions. Program structures can be viewed in execution by users of the system, which has many elements of program visualisation embedded in it.

A large amount of the work carried out on static parallel visual programming systems has been concerned with dataflow or functional models of parallelism. Many of these systems use visual interconnections for the specification of data dependencies or pathways (for example in dataflow program graphs or in CSP style inter-process communication pathways). Text is then used for the behavioral specification of individual components. This approach is often justified by claiming that text is a good communicator of sequential algorithms, and that the structural aspects of parallelism are better represented visually.

Other, mainly Petri-net based, systems visually model the flow of control in parallel systems. Systems based on flow chart like design methodologies with added parallelism have also been proposed. Visualisation of inter-process communication is not usually included in these systems. Again, textual specification is often used as an expression of the functionality of individual components, while graphics are used for high level overviews.

In general, parallel visual programming systems tend therefore to concentrate on particular aspects of the programming paradigm being visualised. Few systems incorporate control flow and inter-process communication in the same model.

2.4.1.1 Petri-net based systems

Petri-nets (Peterson, 1977) are a naturally parallel and visual methodology. They have been widely used for the design and analysis of parallel systems and have had a major impact on parallel visual programming systems. Of the few fully parallel visual programming systems, a large percentage have been based on Petri-net models.

The PFG system (Stotts, 1988) uses hierarchical graphs for the expression of data structures. Enhanced Petri-nets express the flow of control within the system and operations on data structures defined by the graphs.

EDDA (Kerner and Rainel, 1986) is more loosely based on Petri-net theory and makes use of dataflow techniques. Program graphs in EDDA are dataflow in nature, but borrow concepts from Petri-nets such as token generation and the "firing" of nodes. Nodes in the graphs correspond to user defined computations expressed in textual program code, further graphs, or to system defined flow directing "actors".

Coloured Petri-nets (Peterson, 1980) form the basis of the MOPS² system (Ae, Yamashita, Cuhna and Matsumoto, 1986) which is designed to allow parallel systems to be constructed and simulated. Coloured Petri-nets expand on the simple Petri-net model by the introduction of coloured tokens to represent the activation of reentrant code (e.g. recursion). Specifications for modules in the system are written in a sequential, textual, programming language.

A later system along similar lines but aimed at the rapid prototyping of real-time software has also been developed (Ae and Aibara, 1987).

Transaction networks, based on Petri-net theory, are the subject of work by Kimura (1988). Kimura's system supports a produce/consume paradigm in which transaction networks graphically specify computations on "source" and "target" databases. Transactions are atomic actions void of internal state. When transactions are fired they consume data from source databases, leaving their results in target databases. The networks are equivalent to Petri-nets where tokens, places and transactions correspond to data, databases and transactions respectively.

In the SPECS project (Dahler, Gerber, Gisiger and Kundig, 1988) extended Petri-nets are used to model the hierarchical structure and control flow of concurrent systems. Data structures and sequential program behaviour are described using the smalltalk (textual) language.

VERDI (Graf, 1987) is a system intended for the design and specification of distributed computer systems. N-party interactions are described in the language by Petri-net like diagrams labelled with interaction points. Control flow therefore forms the major part of the visualisation, which is animated to allow users to see the program running by watching the movement of tokens around the diagrams.

2.4.1.2 Dataflow and functionally based visual programming systems

Dataflow graphs are another naturally parallel paradigm that has found use in the development of static visual programming systems. Various authors have asserted that functional style dataflow programming languages are far more clearly represented visually than they are textually, for example (Davis and Keller, 1982; Cox and Pietrzykowski, 1985; Gillett and Kimura, 1986a). Most of the dataflow based visual programming languages that have been developed do not produce code for parallel machines. Instead, they compile dataflow languages into code for execution on conventional, sequential machines. As such systems could easily be modified to generate code for parallel computers like the Manchester dataflow machine (Gurd, Kirkham and Bohm, 1987) several systems which produce code only for sequential machines are discussed here.

The dataflow paradigm allows programs to be exclusively represented using graphs with no need for textual descriptions, although some dataflow based visual programming systems do use text to specify the functionality of graph nodes. The use of visual dataflow program graphs has been discussed from a practical and

theoretical viewpoint in (Davis and Keller, 1982) and (Eisenbach, McLoughlin and Sadler, 1989).

PROGRAPH (Cox and Pietrzykowski, 1985) is a visual programming system based on functional programming and the dataflow principle. Data flows along "wires" connecting primitive or user defined processes. User defined processes are built up from primitive processes or further lower level user defined processes. User defined processes at the lowest level are built up entirely from primitive atomic processes. The system incorporates a useful debugger in which the data values on PROGRAPH's wires may be read using a cursor. Figure 2.1 shows an example function from a PROGRAPH program. PROGRAPH is chosen for such a visual example because it provides a good example of the visual dataflow programming style and exhibits many features used by later systems.

A system similar to PROGRAPH called "Show and Tell" is described in (Gillett and Kimura, 1986a). The system relies heavily on the ideas of dataflow, as does PROGRAPH, but allows the use of icons as visual labels for user defined functions, thus adding an extra layer of visual information to the system.

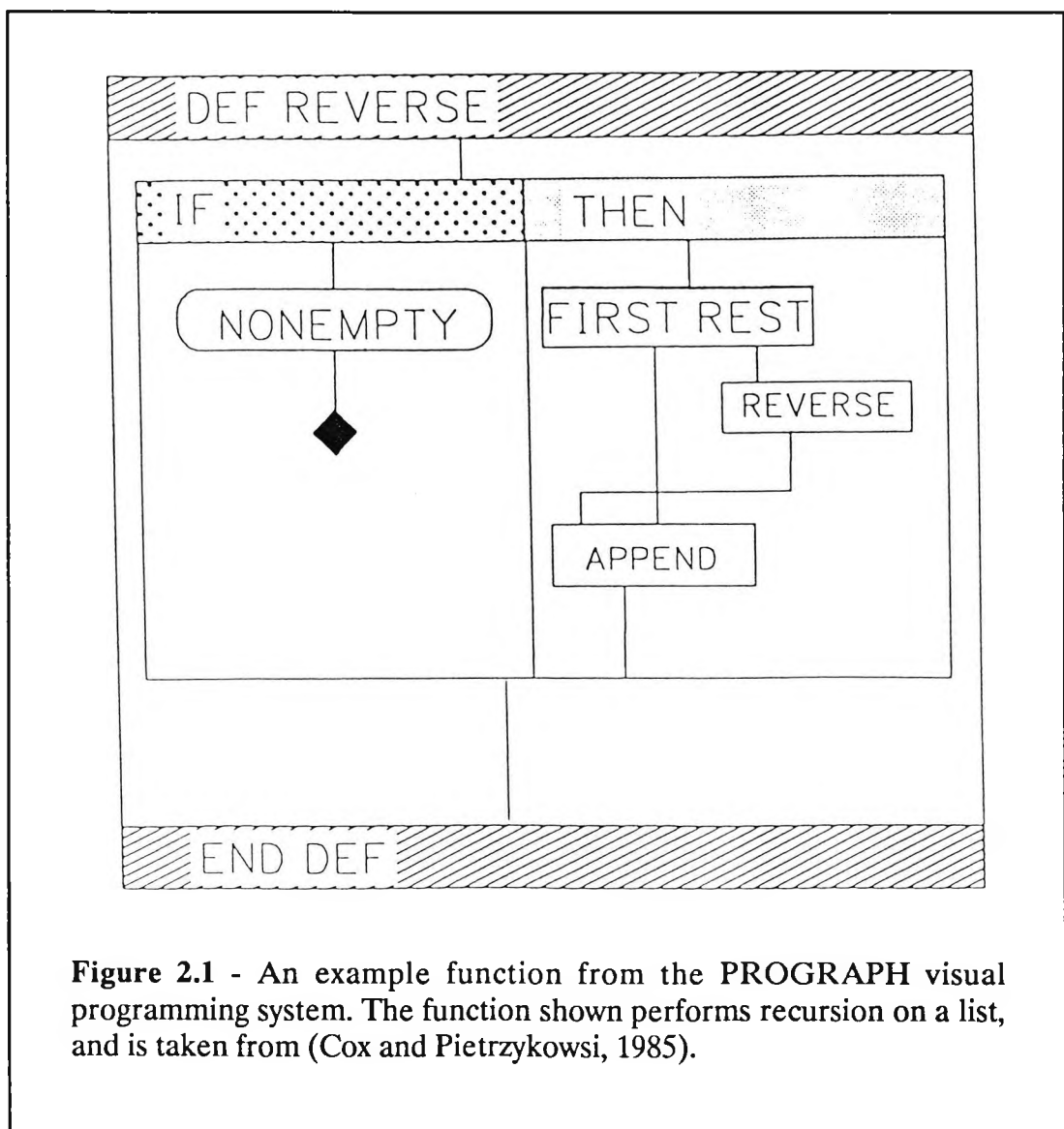


Figure 2.1 - An example function from the PROGRAPH visual programming system. The function shown performs recursion on a list, and is taken from (Cox and Pietrzykowski, 1985).

Two other dataflow based visual programming systems are Fabrik (Ludolph, Chow, Ingalls, Wallace and Doyle, 1988) and Conman (Haeberli, 1988). Both of these systems rely on dataflow "wires" carrying data, but include facilities for interaction with bit mapped displays in their primitive elements. Fabrik is intended for use in introductory programming, while Conman is designed for use with professional graphics workstations.

A dataflow system radically different from those described above is ALEX (Kozen, Teitelbaum, Chen, Field, Pugh and Zander, 1987). ALEX is closest to a conventional dataflow language with added recursion. It allows the graphical representation of two types of objects: data objects and functions. Data objects are represented by rectangles of various sizes, functions or programs by tree-like hierarchical structures. Programming is performed by creating or copying data objects or functions and placing them on the screen. Data dependencies are then specified between the objects by the use of colour coding, not by "wiring diagrams" as in the previously discussed systems.

2.4.1.3 Coarse grain dataflow based visual programming systems

Dataflow systems like those described in the previous section can be regarded as having communicating processes with a very fine grain. Dataflow design methods have also been used as an underlying paradigm for systems which are intended to deal with computations of a much coarser grain. These systems do not implement dataflow or functional programming models directly. Instead they apply dataflow design principles to the specification of inter-process communication. Typically a hierarchical set of dataflow diagrams with atomic leaf nodes completely defines the flow of data within the system. A large number of visual programming systems have been produced around such methodologies, possibly because of their common usage in many areas of the computing industry. Data driven design methodologies are discussed in (Gane and Sarson, 1979).

It should be noted that many of the parallel visual programming systems discussed in other sections of this chapter contain elements of dataflow based design. As exchange of data between computational elements is found in nearly all parallel systems this is possibly to be expected, but only systems claiming to be based on specific methodologies are discussed in this section.

An environment that uses dataflow based design methods is described by (Fisher, 1988). The environment uses dataflow design techniques to specify communication between user defined mixed language modules. It is targeted at developers of scientific and engineering applications, with particular emphasis on the needs of researchers in graphics and image processing.

CAEDE (Buhr, Karam, Hayes and Woodside, 1989) is a visual programming system for the development of ADA applications. CAEDE's diagramming notation is intended for use with a data driven design methodology. Bubbles in dataflow diagrams become boxes in CAEDE diagrams with inter-box connections enforcing desired dataflow patterns. Tools are reported to be in development for

the production of Ada code templates from the diagrams, as well as reverse engineering tools to extract CAEDE diagrams from existing Ada programs.

2.4.1.4 CSP and CCS based visual programming systems

CSP and CCS like methodologies offer significant advantages for parallel processing. The firm theoretical foundation offered by such systems provides potential aid for complex problems such as deadlocks. In addition, CSP's static networks of communicating processes are well suited to the application of current visual programming techniques. Given this ease of application it is surprising that more visual programming systems based on the methodology have not been developed. Visual programming techniques have the potential to offer users better interfaces to such methodologies and languages based upon them. The application of visual programming techniques to a CSP style programming system and methods for the development of systems supporting such a methodology is the subject of the remainder of this thesis. Some details on this work have already been reported (Roberts and Samwell, 1989), (Roberts and Samwell, 1990). This work is not reviewed here, but is included in appendix three.

An early CSP style visual programming environment using a mixed text and graphics approach is reported in (Pong, 1986). Programs are constructed from boxes representing processes and containing sequential code written in a Pascal like language with extensions for inter-process communication. Construction of sequential code defining the behaviour of processes is aided by a textual code editing environment that supports the use of Nassi-Shneiderman diagrams. Links between the icons are drawn to define communication pathways corresponding to Occam channels, thus specifying inter-process communication. The system produces code for execution on a simulated ring of processors. Figure 2.2 shows an example inter-process communications diagram from Pong's system, IPIGS, included because IPIGS was the first CSP based visual programming system.

An almost identical approach for Transputer based Occam programs is taken in (West and Capon, 1990), which does not include a Nassi-Shneiderman diagram editor but uses similar techniques to aid the automatic generation of procedure headers and process "harnesses" for Occam programs. Neither system restricts inter-process communication patterns in any way.

A formally based visual programming system based on a two dimensional representation of the CCS formalism (Milner, 1980) has been developed for the Clara environment (Giacalone and Smolka, 1988). Clara is aimed at supporting CCS for the specification, development and simulation of a wide range of parallel systems. Graphical CCS style processes are constructed by the "gluing together" of a range of textually labelled icons which correspond to CCS constructs. User constructed processes may be abstracted to form processes that may in turn be used in the construction of further expressions. The visualisation used has been found to be extremely helpful in aiding the use of CCS over a range of parallel systems. Figure 2.3 shows an example Clara graphical representation of a CCS construct. An example from Clara is included because it is the only CCS based

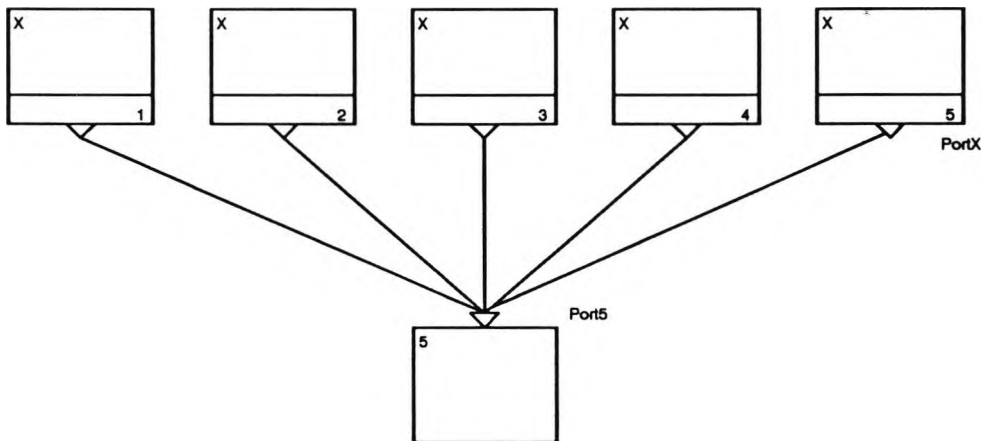


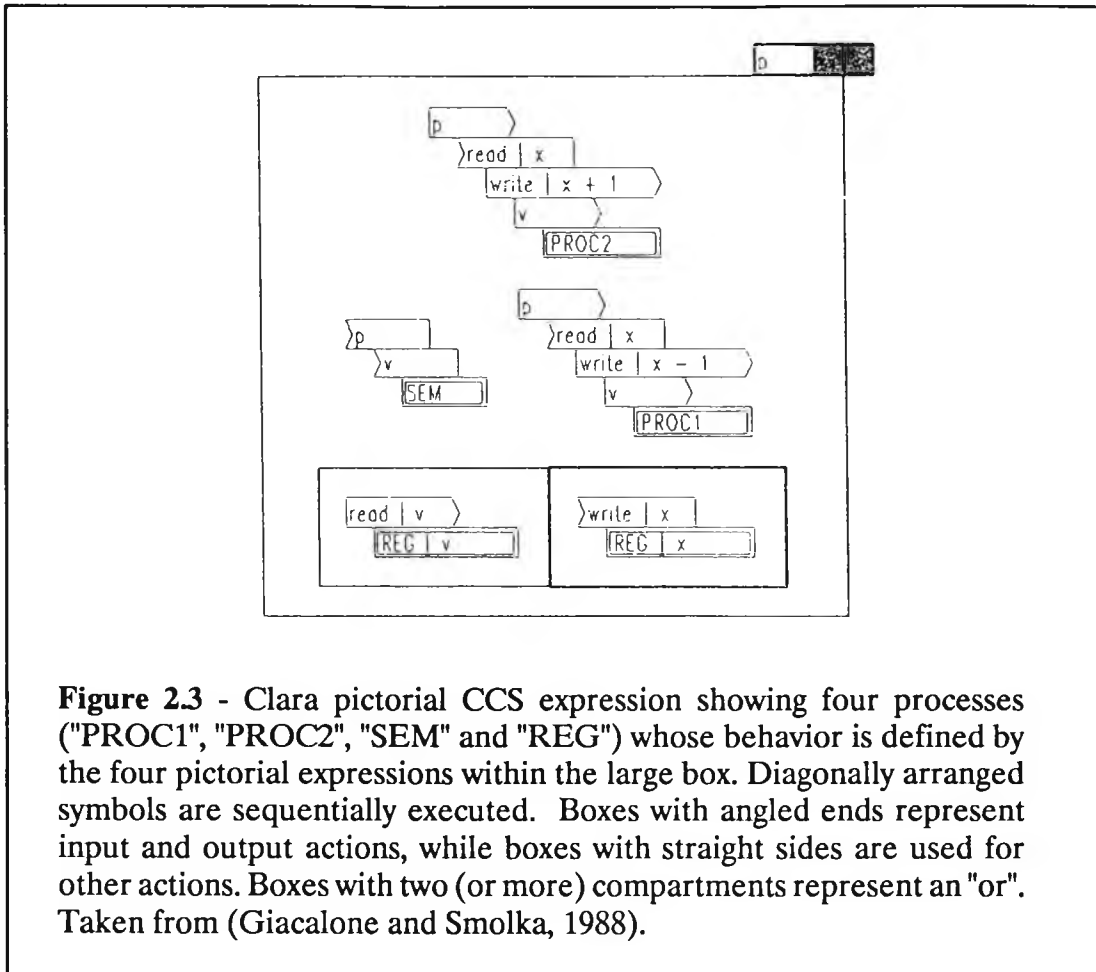
Figure 2.2 - An example inter-process communication diagram from the IPIGS system which shows a single process communicating with an array of five processes. The figure is taken from (Pong, 1986).

visual programming system in existence and because it is one of a small number of systems reviewed here which are based on formally well founded principles.

2.4.1.5 Other visual programming systems

As mentioned earlier, most other parallel visual programming systems may be regarded as containing elements of dataflow design, since the flow of data between computation elements is a universal property of parallel computations. They range from systems designed for the production of language and architecture independent parallel programming to highly machine specific tools aimed at machine level programming.

In addition to the parallel visual programming systems mentioned above, brief reviews of some sequential systems using communication based methodologies suitable for use in parallel systems are included in this section.



HI-VISUAL (Mondon, Yoshino, Hirakawa, Tanaka and Ichikawa, 1984) is a sequential visual programming system targeted at the image processing area. Users of HI-VISUAL select from a large number of pre-defined icons and connect them into networks to produce applications programs. Icons are regarded as functions which have inputs and outputs, with programs constructed by the specification of connections between them. It is easy to see how such a system could take advantage of a parallel machine by mapping the processes represented by icons to physical processors. Figure 2.4 shows an example HI-VISUAL applications program for detecting cracks in printed circuit boards, included because HI-VISUAL is a well known iconic visual programming system.

Poker (Snyder, 1984) is a system designed for performing programming on the Cosmic cube computer and parallel machines based on the ChiP microprocessor. Like many other systems, Poker uses a visual programming language for the expression of inter-process communication structures, while code for processes is written using a Pascal like language with inter-process communication extensions. Graphs describing the communications aspects of programs are produced by drawing connections between boxes representing processors arranged in a grid. Code is then written for the processors on a separate display using a text editor. Inter-process communication in Poker appears to be data driven to the programmer, but is mapped automatically into the synchronous systolic communication facilities of the target microprocessors.

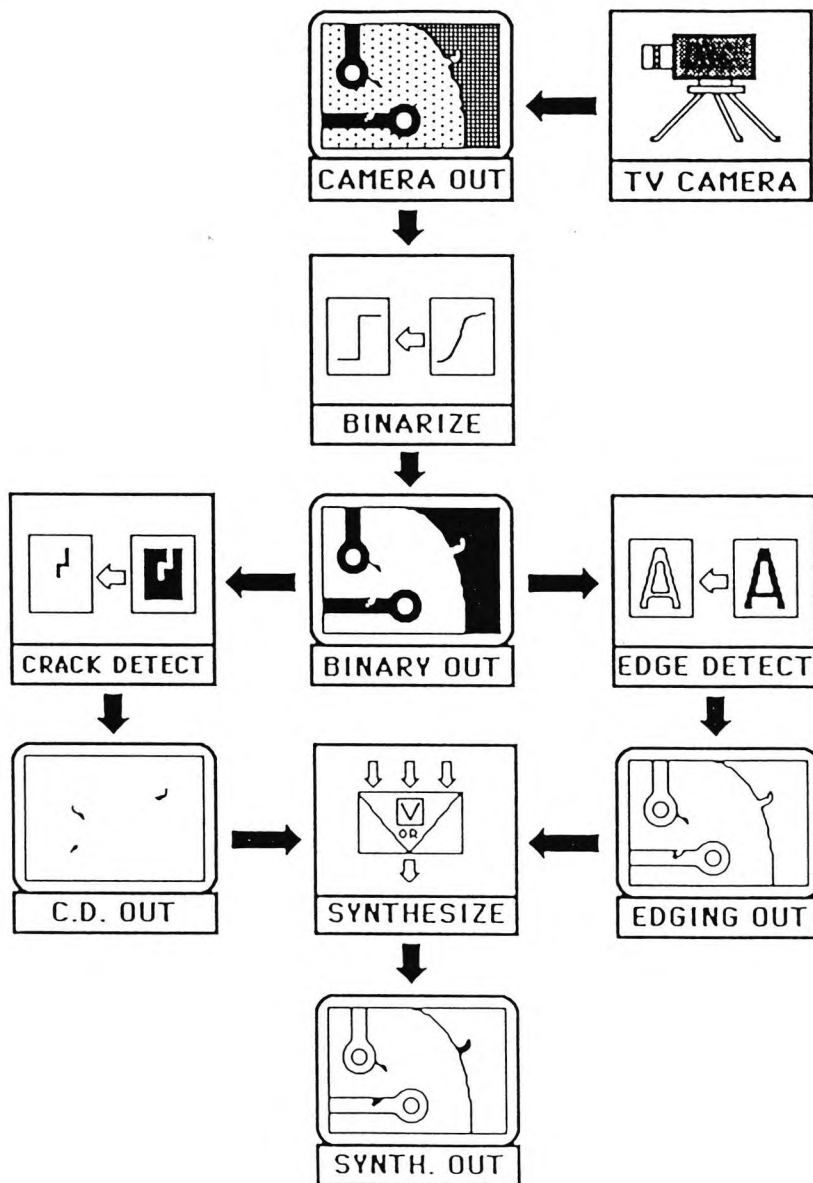


Figure 2.4 - An example application program in HI-VISUAL. The program detects cracks in printed circuit boards using image processing techniques on a captured image. Taken from (Shu, 1988)

Systolic programs for such microprocessor arrays typically have fixed (static) inter-process communication structures which are easy to express visually. Processes execute simple algorithms well suited to expression using imperative textual programming languages. The area is ideally suited for visual programming techniques. A version of Poker specialised to just this type of computation has also been produced (Snyder, 1987).

STILE (Stovsky and Weide, 1987) and the Conic environment (Kramer, Magee and Ng, 1989) are two parallel visual programming systems similar to Poker but, unlike Poker, both allow hierarchical program structures to be visually evolved. Programs in such systems consist of process "boxes" interconnected by communication links. Boxes can contain further graphs or, at the lowest level of abstraction, contain simple textual program fragments. STILE is aimed primarily at real-time programming applications and the problems involved in real-time process control, while the Conic environment is more generally aimed at distributed and concurrent programming. Of the two systems, the Conic environment has more advanced editing facilities. Kramer, Magee and Ng assert that both STILE and Conic are based on the principles of "Configuration Programming". Configuration Programming is closely related to the ideas of programming in the large and module interconnection languages (DeRemer and Kron, 1976). In such systems, graphical connections between "modules" define communications structure and system "configuration" in a graphical way, while textual specifications or further graphical configurations define modules at lower levels of abstraction. Many mixed graphical and textual programming languages may therefore potentially be termed configuration languages. It should be noted that such graphical configuration languages in general have little to do with Occam's configuration statements (May, 1987), which are concerned with the placement of processes upon processors. Both STILE and the Conic environment do however provide the facility to affine processes to particular processors in distributed systems.

A very highly machine specific visual programming language has been produced for the Navier-Stokes computer (Tomboulin, Crockett and Middleton, 1988), which is a high-performance, reconfigurable, pipelined machine designed for solving computational fluid dynamics problems. Inefficiencies in mapping conventional high-level languages to the Navier-Stokes' architecture constrains programming on the machine to a very low level. To produce efficient code, programmers must have express knowledge of the machine's architecture. The visual programming system has been shown to provide significant complexity-management advantages over the textual microcode methods previously used and is able to aid programmers by managing a large amount of previously manually performed tasks.

In contrast, CODE (Sobek, Azam and Brown, 1988) is a system aimed at producing parallel programs that are portable across a wide range of MIMD target architectures. CODE programming is performed at a very high level of abstraction using graphically represented data dependencies, units of computation and filters. These elements are combined in graphs which may contain sub-graphs. After a CODE program has been drawn, a description file is transferred to the target architecture. A compiler running on the target architecture then compiles the description file for execution by a simple run time system. Recoding the run time system for different architectures is claimed to involve little work.

In (Bhattacharyya, Cohrs and Miller, 1988) a visual interface to the inter-process communications facilities of Unix is described. The system allows inter-process connections to be graphically specified. Definitions for the processes are written

in standard Unix fashion using C or shell scripts. Taskmaster (Arthur and Raghu, 1989) provides similar facilities with added hierarchical process structuring, monitoring and debugging facilities. A similar, earlier and less extensive system has also been produced for the Lilith/Modula computer (DeMarco and Soceneantu, 1984). All these systems generate code for conventional machines, but it is not hard to image such tools in use with multicomputer operating systems like Meshix (Winterbottom and Osmon, 1990) or Helios (Perihelion, 1988).

2.4.2 Graphical CASE tools and visual simulation systems

A number of CASE tools produced for use with specific systems design methodologies are also worthy of mention here.

A description of possibly the most relevant system in this category may be found in (Crowe, Hasson and Strain-Clarke, 1989). Program specifications are constructed using "bubble and arc" diagrams in which bubbles correspond to processes and arcs to inter-process communication. Bubbles contain further diagrams or, at the lowest levels of abstraction, they contain textual specifications written in a form of CSP. The system is intended to work as a programmer's aid in the production of deadlock-free Occam programs by allowing the programmer to formally reason with CSP in a highly humanistic manner. Crowe, Hasson and Strain-Clarke's tool is essentially a dataflow based design tool with an inherent highly specific definition of communication between primitive elements in the diagrams.

Other environments aim to support system design methodologies such as MASCOT (Simpson, 1986) and SDL (CCITT, 1984) in a graphical way.

A design support environment has been produced for MASCOT (Looney, 1988). The environment allows graphical input of MASCOT diagrams. Designs produced using the system may be simulated and advisory statistics produced. Similar environments for SDL have also been produced (Koyamada and Shigo, 1988; Orr, Norris, Tinker and Rouch, 1988; Nakamura, Fujimoto, Suzuki, Tarui and Kiyokane, 1986). All these systems have been targeted at the verification and design of communication protocols and software for use in the telecommunications industry. Another system in the same area, PROSPEC (Chow and Lam, 1988), applies communicating finite state machines to the design and verification of communications protocols. All these tools rely heavily on visual programming techniques in the specification of their input diagrams, yet are clearly CASE tools.

Graphical CASE systems share many common features with systems for the visual simulation of parallel architectures and software systems.

PARET (Nichols and Edmark, 1988) uses behavioral simulation to predict the interactions between elements of a simulated parallel system and to allow the effect of software systems on parallel architectures to be analysed. Architectures may also be analysed separately from software systems. Graphs representing

systems are interactively input to the system which then graphically displays simulation results on a workstation screen using dynamic information overlay.

Starlite (Cook and Auletta, 1986) is a visual simulation system for the prototyping of distributed applications. Although textual specifications are used to define simulated systems, the system provides extensive facilities for graphical display of simulation results. Displays take the form of system defined visualisations of items like ethernet and clocks, with simulation specific information overlaid on them.

Also relevant is the Gecko system (Stephenson and Boudillet, 1988), which is discussed later in this chapter.

2.4.3 Program visualisation systems for parallel machines

The aim of program visualisation systems is to help programmers form clear and correct mental images of a program's structure and function (Brown, Carling, Herot, Kramlich and Souza, 1985). They use graphics to illustrate aspects of a program's behaviour and design as a means to achieve this goal. A cross section of program visualisation systems for sequential machines has been reviewed in (Myers, 1988) and a selection of recent program visualisation systems intended for the debugging of parallel and distributed systems can be found in (LeBlanc and Miller, 1988). General issues concerning the monitoring of distributed systems and the graphical display of information obtained through monitoring is discussed in (Joyce, Lomow, Slind and Unger, 1987), while (Pancake and Utter, 1989) reviews recent work on visually based parallel debuggers and contrasts visualisation approaches with underlying computational models.

It should be noted that, while it is possible to "re-visualise" programs produced by visual programming systems, most program visualisation systems work with conventional, textual languages.

Graphical performance monitoring systems use graphics to display statistics on a computer's performance. Typically statistics such as processor loading and utilisation of communications subsystems are shown. Many performance monitoring systems use program visualisation to relate performance information to areas of program code with the intention of revealing performance bottlenecks within the code. These systems will be referred to as program level performance evaluation tools and are discussed jointly with other related program visualisation systems. System level performance analysis tools, producing views of overall system performance, are not discussed here. The reader is directed to (Tang, 1988) for further details on this subject.

Program visualisation systems for parallel computers may be divided into two classes; static visualisations and dynamic visualisations. Static visualisations show constant views of target programs, which do not change with time. Typically these visualisations reflect aspects of programs which are not time dependent, such as data dependencies, fixed inter-process communication links or non-dynamic control flow structures. Dynamic visualisations reflect changing aspects of

programs, such as usage of data areas. Some dynamic systems calculate static views of programs under consideration, overlaying them with information such as performance data or shading, perhaps representing executing concurrent processes. Other systems recalculate or iteratively modify views of the program as execution proceeds.

Although it appears at first consideration that dynamic systems are potentially more useful than static systems, the extraction of data from concurrent computers has a potentially perturbing effect on the execution of the program being monitored. This so called "probe effect" is caused by the fact that any software based extraction of data from a monitored machine frequently causes delays in execution in the system being monitored. Any such delays in a non-deterministic program may result in a change of system behaviour and hence differences between output from monitored and unmonitored systems may be detected. A cross section of methods for the extraction of information from parallel systems can again be found in (Joyce, Lomow, Slind and Unger, 1987), while (LeBlanc and Miller, 1988) also contains details on relevant work.

2.4.3.1 Static program visualisation systems

The majority of parallel program visualisation systems display static views of programs, representing facets of their execution behaviour. Two systems which are typical of static program visualisation systems are ART (McDowell, 1988) and GRAIL (Stepney, 1987). Both ART and GRAIL carry out static program visualisation with static overlay of information.

In ART, static analysis is carried out on parallel programs to reveal synchronisation and data usage errors. Programs are displayed as synchronisation graphs (similar to flowcharts) upon which information about potential errors is overlaid. The ART system typifies the static analysis approach to the visualisation of parallel programs in which program source code is analysed to indicate potential problem areas and results are graphically displayed. The analysis of the program is carried out either according to a set of heuristics, or to some underlying theoretical background.

GRAIL is a program level performance analysis system for "T-rack" Transputer systems running Occam programs. It uses a common approach in which a static visualisation of a program's structure is overlaid with performance information obtained through the monitoring of the system. The structure of Occam processes and channels is displayed by GRAIL, which also displays the Transputers on which the processes reside. All information is drawn on the screen of a colour Sun workstation. Information produced by extra Transputer instructions inserted into compiled Occam programs by a specially modified compiler is used to colour the visualisation. The colours provide a picture of where the program spends most of its time, the utilisation of channels and the utilisation of inter-processor links. Code where the processor spends a lot of time is coloured red (for "hot") while little used code is coloured blue (for "cold"). Suitable shades between the two extremes indicate different levels of usage. A similar scheme applies to the

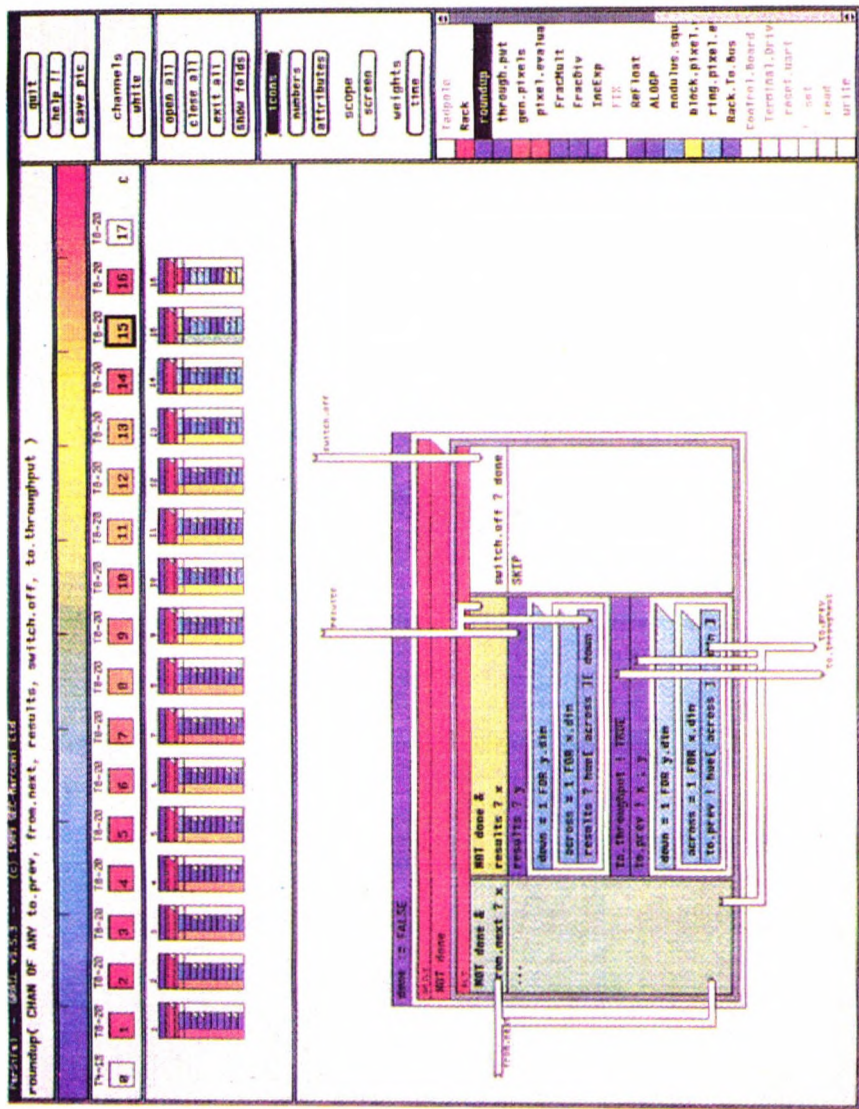


Figure 2.5 - A screen from GRAIL showing an Occam process and associated channel connections. Taken from (Hughes, 1989).

processor utilisation, link and channel usage statistics. Figure 2.5 shows an example screen from GRAIL, included because GRAIL has proved to be an invaluable asset for the performance debugging of T-rack Transputer arrays, is widely used where it is available, and is possibly one of the best examples of a program visualisation system in common usage. A version of GRAIL which extracts information directly from executing code on Transputer arrays has also been suggested, though it is yet to be implemented.

2.4.3.2 Dynamic program visualisation systems

Many dynamic program visualisation systems generate static views of programs which are then overlaid with changing information (for example, tokens moving around a network to indicate control flow or changing levels of processor utilisation). These systems will be referred to as having static visualisations with dynamic overlay of information. Other systems iteratively modify program views, or calculate new views as required. These systems will be termed truly dynamic program visualisation systems or, in shorthand, dynamic program visualisation systems.

Often, the decision to use a static visualisation with dynamic overlay or a truly dynamic visualisation is related to the underlying paradigms used by the programming system and the facet of execution which it is desired to display. For example, truly dynamic visualisation techniques are highly appropriate for viewing processes in systems with constantly evolving sets of processes, e.g. a Unix style system. Static systems with dynamic overlay are more appropriate for viewing properties of processes in fixed process networks.

2.4.3.2.1 Static program visualisation with dynamic overlay of information

Animated debugging of ADA programs is the subject of work reported in (Moran and Feldman, 1985), which uses static visualisation with dynamic overlay. Ada tasks are graphically represented as rectangles with unique names and colours. Entry queues for tasks are represented by labelled rectangular boxes justified down the left hand side of particular tasks. Tokens, coloured in the colour of the calling task, are deposited in the entry queues where they wait until accepted. When rendezvous are performed, calling and accepting tasks blink and stocks of tokens are appropriately depleted. Similar techniques are used by (Garcia and Berman, 1985) who present work on the animated debugging of programs written in a parallel version of Pascal. Programs are visualised as Petri-nets. Tokens move around the nets to indicate the execution of constructs and other facets of the program's execution.

A novel system for debugging Occam programs using "Occam control graphs" (a parallel version of flowcharts, with added inter-process communication) has been reported recently (Mourlin and Cournarie, 1990). Occam programs are transformed into the graphs, which are displayed on a workstation screen. Location

of control in the system is shown by darkened nodes in the graphs. Graphs can be viewed at any desired level of abstraction, while separate windows show traced variables and program code. The system allows both static and dynamic analysis of program code, with the animation features allowing the detection of deadlocks, livelocks and other similar errors. Information for the animations is obtained from a multiprocessor simulator, thus avoiding probe effect. The authors claim that the structure of the Occam language is particularly well suited to graphical representation and that graphical environments are particularly valuable in helping understand how parallel programs really work. The system at present supports only the Occam1 (Inmos, 1984) language, but its authors plan to expand it to support the newer Occam2 (May, 1987).

2.4.3.2.2 Truly dynamic program visualisation systems

Voyer (Bailey, Socha and Notkin, 1988) is a dynamic system for constructing application specific animated views of parallel programs. Instead of providing a uniform visualisation system for different pieces of software, Voyer's designers take the approach that the program developers should annotate their programs to send appropriate messages to data collection subsystems. As well as annotating their programs, program developers also write a "modeller" and a "renderer" to rationalise and display information. Voyer handles all system level interaction, debugging, message routing, etc., leaving the user to write the high level routines already mentioned. Voyer has been implemented for the Poker programming system (described earlier) and on a Sequent Balance. Various visualisations of parallel algorithms have been programmed using the system.

Tree (Tang, 1987) is another truly dynamic system which provides its user with an animated graphical view of events which represents a program in execution on the Manchester Dataflow machine. Tree shows executing processes as boxes. As new processes are spawned by the machine to evaluate expressions, progressively more boxes are added and linked to those already in existence to show their ancestry. Processes suspended in execution are shown in reverse video. This simple visualisation allows the systems "throttle" mechanism (which matches parallelism in algorithms to available parallelism in the machine) to be viewed at work. It facilitates evaluation of the performance of the machine, but is of limited use during debugging.

MONA (Joyce and Unger, 1985) is a dynamic tool for the animation of message interactions between concurrent processes over loosely coupled distributed Unix systems. It visualises message interactions between concurrently executing processes by representing processes as labelled circles. Requests for communication and operations like data reception are shown using a variety of simple line styles and highlighting operations. The authors report that, although simple, the system shows that graphical monitoring tools such as Mona are immensely helpful in obtaining global pictures of the operation of distributed systems.

A relational approach to the visualisation of parallel programs is taken in (Schwan and Matthews, 1986). Relationships between components of parallel programs are entered into a relational database, later to be visualised on a screen. As the system concentrates on the storage and display of information relationships have to be manually entered. It does not include features to extract data from programs or from processors. Once relationships have been entered, the user interacts with the systems to produce views of the program under consideration which show desired features. The result is a highly flexible program visualisation system in which users are able to specify view contents. Combined with a flexible monitoring system performing continuous updating of the database such a system could provide a highly flexible dynamic program visualisation tool.

(Bemmerl, 1988) discusses a variety of tools for use with parallel systems, including a dynamic visualisation tool. The tool shows an animated display of the flow of communication between processes, but is reported at an early stage of development and thus few details are given.

IDD (Harter, Heimbigner and King, 1985) is a system for debugging distributed systems using a form of temporal logic. Modified temporal logic expressions constrain the information collected by the monitoring system. This information (specifically inter-process communication) is displayed on a graphics screen as lines drawn between points which represent processes. The position of the points is determined by a number unique to each process in the system (y axis) and by the time at which each process commits to the communication process (x-axis). Diagrams are updated as new information arrives at the monitoring system. The approach taken is similar to the concurrency maps and time line diagrams described in (Stone, 1989).

PIE (Lehr, Segall, Vrsalovic, Caplan, Chung and Fineman, 1989) is a mature visual performance debugging system, having been in existence for several years. Like GRAIL, discussed earlier, PIE displays information to aid the improvements in the performance of parallel software running on a number of machines. Data is collected from programs under study using software "sensors". A variety of views including dynamic visualisations of process spawning and oscilloscope like traces of process execution are supported by the system. PIE's authors feel that PIE is not only a good tool for performance debugging, but also a useful one for "understanding the complexities of sequential and parallel programming".

GRADIVAL (Vornberger and Zeppenfeld, 1990) provides similar facilities to those provided by PIE, but for Transputer systems running Occam programs.

2.5 Visual process to processor mapping tools

Gecko (Stephenson and Boudillet, 1988) is an interesting system which can display the results from a simulation of a distributed application and perform process to processor mapping. Gecko can visualise Occam programs using a simple "bubble and arc" diagram with added colour. Processes may then be mapped onto processors in a Transputer system by the interactive placement of bubbles onto

icons representing processors. Alternatively the system can display output from the a parallel architecture simulator. The system thus contains elements of program visualisation, visual configuration and visual simulation.

Other, simpler, visual tools for the setting up of reconfigurable processor arrays have also been produced. Express (Levco, 1990) includes such a tool for reconfiguring arrays of Transputers, while a more generalised tool for Transputer based systems is also under development (Tilley, 1990).

2.6 Review summary

This chapter has discussed a number of graphical tools related to the work reported in the following chapters of the thesis. Although it is clear that visual programming techniques will not quickly replace conventional textual languages for use in general purpose programming (Myers, 1988) the relatively primitive graphical program development systems reviewed in this chapter have proved that visual programming techniques can be successfully applied to the development of parallel systems. The work of the following chapters draws its inspiration principally from three systems discussed earlier - HI-VISUAL (for the use of iconic process representation), STILE (for visual hierarchy) and GRAIL (for a "mixed" visualisation). It should however be noted that the GILT system is not a development or enhancement of any of above systems, but a completely new system in its own right.

Visualisations of Occam programs and Transputer systems

3.0 Introduction

This chapter introduces relevant features of Occam and the Transputer which provide a basis for subsequent discussion and analysis of existent visualisations of Occam programs and Transputer systems. The GILT language, based on some of these features is introduced together with a discussion on some basic design decisions taken during its development. The whole chapter provides a background for chapter four, which deals with formal representations suitable for the expression of GILT's syntax and semantics, and chapter five, which provides a full definition of the language.

3.1 The Occam programming language

This section provides brief refreshment on features of the Occam (May, 1987) programming language which are relevant to the subject of this thesis. It should not be considered a complete definition of the language. Many language features are omitted, and those that are described are discussed very briefly. Complete information may be found in (Jones and Goldsmith, 1988) and (May, 1987), while a complete syntax may be found in appendix one.

Two flavours of the Occam language have been produced so far. The first, Occam1 or "proto Occam" (Inmos, 1984), is far simpler than the second, Occam2 (May, 1987; Jones and Goldsmith, 1988), which contains more advanced data structuring and other facilities. The Occam referred to in this thesis is Occam2, except in cases where explicit references are made to Occam1.

Occam is a language designed for parallel processing with the aim of exploiting multiple processors connected together using communications links. It is an imperative language with a basis in the CSP (Hoare, 1985) formalism. Two elements are fundamental to Occam - "processes" and "channels". All processes, with two exceptions, start, perform some action, and then terminate. Channels provide a mechanism for communication between processes executing in parallel. Parallel processes are not allowed shared variable access of any type or form.

3.1.1 Data types, variable definitions and constants

Expressions and variables in Occam may have a value of type BOOL, BYTE, INT16, INT32, INT64, INT (length implementation dependent), REAL32 or REAL64.

The most basic variable definition takes the form of a type keyword followed by a list of variable names separated by commas and terminated with a colon. Mechanisms for the specification of arrays of all types are included.

Example

```
INT variable :
```

Variables may be renamed by abbreviations:

```
INT a.cat IS a.siamese :
```

Constants (values) are introduced using the keyword VAL in a similar fashion :

```
VAL INT cats.lifespan IS 17 :
```

3.1.2 Channel protocols

Channels have protocols in the same way that variables have types. Protocols are used in channel definitions. A protocol may be given a name in a protocol definition.

Example

A protocol for a channel carrying strings of known length might be given as follows:

```
PROTOCOL STRING IS INT::[]BYTE :
```

Various other types of protocol definition exist.

3.1.3 Channels

Channels form the medium for communication between processes. Channel definitions are very much the same as variable definitions, but use channel protocols instead of types.

Example

```
CHAN OF STRING string.chan :
```

Arrays of channels are also permitted.

Example

```
[100]CHAN OF STRING rope :
```

3.1.4 Primitive processes

The most basic processes in Occam are the three "primitive" processes; "assignment", "output", and "input":

3.1.4.1 Assignment

In an assignment the value of an expression is assigned to a variable.

Example

```
variable := expression
```

3.1.4.2 Output

The "!" symbol is used for output to a channel. In an output, the value of an expression is output to a channel.

Example

```
channel ! expression
```

3.1.4.3 Input

The "?" symbol is used for input from a channel. In an input a variable receives a value from a channel.

Example

```
channel ? variable
```

"channel" is assumed to be a channel, "expression" an expression, and "variable" a variable.

3.1.5 Scope of channels and variables

Channels and variables must be declared before usage. The scope of such specifications is defined to be the text of the syntactic entity following the specification, excluding any text in the scope of a specification with the same name.

3.1.6 SKIP and STOP processes

Two special processes, "SKIP" and "STOP", perform differently to most processes. SKIP starts, performs no action, then terminates. STOP starts, performs no action, but never terminates. Both SKIP and STOP are supported as keywords in Occam, but they are also used to describe the action of other processes.

3.1.7 Constructed processes

Processes may be combined using constructor processes to form further processes or constructs. Constructs may be nested to any required depth, though particular implementations may set limits on the depth of the nesting. Each constructed process consists of a constructing keyword ("SEQ", "IF", "CASE", "PAR" or "ALT") followed by a number of component processes.

Component processes in a construct are always on separate lines and slightly indented from the position of the constructor process keyword.

Regular arrays of processes can be formed by replication. This is discussed after a brief synopsis of the simple form of each construct.

3.1.7.1 Sequential construct (sequence)

A sequence or sequential process is formed by the keyword SEQ followed on subsequent lines by zero or more component processes. Processes in a sequential process are executed one after another.

Example

```
SEQ
  x := 1
  y := x + 1
```

Sequences may also be constructed by replication. A sequence with no components is equivalent to SKIP.

3.1.7.2 Conditional processes

A conditional process consists of the keyword "IF" written above a slightly indented list of components. Each component is either a further, nested, conditional process or a Boolean expression, below which is a further indented process. The Boolean expression is referred to as the condition.

The construct is executed by "downwards" testing of Boolean conditions until one is found that is true. An appropriate process is then executed. Erroneous conditional processes without any true conditions behave like STOP.

Example

```
IF
  a < 0
    less.than := TRUE
  a = 0
    equals := TRUE
  TRUE
    greater.than := TRUE
```

The case discrimination process behaves in a similar manner but provides a clearer syntax where a conditional process is required to select a branch according to which constant value in a range of constant values is taken by an expression.

Example

```
CASE a
  10, 30
    b := 7
  20
    b := 10
  ELSE
    b := 0
```

Only one default "ELSE" expression is allowed.

Conditional processes may also be constructed by replication (section 3.1.10).

3.1.7.3 Parallel construct

A parallel construct may be formed by the keyword PAR followed on subsequent lines by zero or more component processes. Processes in a parallel construct are executed concurrently.

Example

```
PAR
  x := 0
  y := 0
```

Parallel constructs may also be constructed by replication. A parallel construct with no components is equivalent to SKIP.

A parallel construct is invalid if any of its components may change the value of a variable that may be used in any of its other components.

Only processes which are in separate components of parallel constructs are allowed to communicate using "?" and "!". Inter-process communication in Occam is synchronised and processes only communicate when both processes are ready.

Parallel processes may also be formed by replication and processes may be allocated to particular processors in a network by the use of PLACED PAR. Parallel processes, in particular processors, may also be annotated with directives to indicate the relative priority of their components using PRI PAR.

3.1.7.4 Alternative processes

The alternative process formed by the use of the "ALT" keyword performs like a multi-way conditional except in that the choice of executed processes depends on whether another process is executing an output. It is written as an ALT above an indented list of "guarded processes". In the simplest form of an alternative, each guarded process is an input process, followed by an accompanying indented process.

Example

```
ALT
  up ? increment
    x := x + increment
  down ? decrement
    x := x - decrement
  read ? request
    reply ! x
```

With more complex guards, some input processes may be preceded by Boolean expressions, preventing their guarded process from being executed under certain conditions. Other variations are possible, including the use of the SKIP process for a default guard.

3.1.8 Comments

Comments are introduced using a pair of dashes ("--") and extend to the end of the line they are introduced on.

3.1.9 Procedural abstraction

Procedure definitions "name" processes and form part of Occam's abstraction mechanism.

Example

In the scope of the procedure definition :

```
PROC do.nothing()  
  SKIP  
:
```

the "call"

```
do.nothing()
```

may be used to stand for the body of the procedure i.e. :

```
SKIP
```

(Whether anyone would desire such a lengthy abbreviation for a process which performs no action is debatable!).

Channels and variables may be passed as parameters to procedures, using a similar declaration to that used in the earlier channel and variable declarations.

3.1.10 Loops and replicators

Two types of loop are allowed in Occam, WHILE loops and indexed FOR loops. While loops are necessarily always sequential, while indexed FOR loops need not be. Indexed FOR loops with constant bounds are used as replicators as mentioned earlier.

Example

The loop

```
WHILE TRUE  
  do.nothing()
```

continues to perform no action forever.

While the loop

```
PAR i = 0 FOR 100
  do.nothing()
```

executes 100 SKIP statements in parallel.

3.1.11 An example Occam program

A simple Occam program fragment for a buffering system is shown in figure 3.1.

```
[5]CHAN OF INT link :
-- definition of single buffer process
PROC buffer.1(CHAN OF INT in, out)
  WHILE TRUE
    INT x :
    SEQ
      in ? x
      out ! x
  :

-- Main program
PAR
  buffer.1(to.queue, link[0])-- First buffer
  PAR i = 0 FOR 4
    buffer.1(link[i], link[i+1]) -- buffers
  buffer.1(link[4], from.queue) -- Final buffer
```

Figure 3.1 - An example Occam program, demonstrating the use of buffering process "buffer.1" in a queuing system. The existence of channels "to.queue" and "from.queue" is assumed.

Integers enter the queue of figure 3.1 via the channel to.queue. The first "buffer.1" process reads from "to.queue" and outputs to the first element of the array of channels "link[0]".

Subsequent instances of the "buffer.1" process read from one channel "link" and output to a second. The second channel has an index of one higher than the first. Integers are passed down the chain of processes via the channels until they are finally output by the final "buffer.1" process onto the channel "from.queue".

In essence, the Occam model of computation restricts computation to be inside a process and allows inter-process communication through the use of channels only. All data and state information belonging to a process is maintained within the process's own private address space and is accessible only to that process. Programs may however be coded to allow processes to request data via a message on a channel. Data may then be returned along a channel as a further message.

3.2 The Transputer

This section provides a brief tour of relevant features of the Transputer. Further information may be found in (Inmos, 1989a).

The Transputer, which is a physical realisation of the Occam computational model, has been designed as a building block for concurrent multi-processor computers. The Transputer family consists of a number of single chip devices. Each Transputer in the family contains a processor, memory, and four serial communication links for connection to other Transputers. Additional memory and links may be externally connected as required.

Transputer systems generally consist of a network of Transputers connected via the previously mentioned communications links. Transputers do not take the Occam computational model down to VLSI level. Instead, each Transputer may execute a number of tasks and, with the aid of a microcoded scheduler, simulate multiprocessor concurrency. Transputers are able to execute a number of low-level processes such as i/o to or from links and arithmetic operations simultaneously due to a certain amount of parallelism within the processor itself. A queue of active processes which are ready to be executed is maintained by the scheduler. Time-slicing is used to execute queued processes and simulate parallel execution on a single processor. It is thus possible to run parallel Occam programs using simulated concurrency on a single Transputer.

From a pragmatic point of view, any program running on a network of Transputers consists of a number of mostly sequential processes. Parallelism is therefore coarse grain.

Providing problems can be suitably decomposed, the overall processing rate of the Transputer array is limited only by the program's logical constraints and the number of processors available in the system.

3.3 Configuration of Occam programs for execution on Transputer systems

To run an Occam program across a number of processors it is necessary to allocate processes to Transputers and channels to physical links. The process to processor mapping function is statically performed before runtime at the program level using the Occam language extensions **PLACED PAR** and **PROCESSOR n**, where **n** is a logical identifier called the processor number. The processor number uniquely identifies a processor in the Transputer network. The entire mapping process is known as configuration.

Example

```
PLACED PAR
  PROCESSOR 0
    do.nothing()
```

Indicates that processor number 0 is to run the process `do.nothing()`. The procedure `do.nothing` is assumed to be a suitably compiled procedure.

An extension used in the Inmos Transputer development system (Inmos, 1988) specifies which member of the Transputer family the processor belongs to. It takes the form of an additional alphanumeric specification after the processor number in the **PROCESSOR** statement.

Example

```
PLACED PAR
  PROCESSOR 1 T8
    do.nothing()
```

This indicates that processor number 1 is a member of the T8 family (for example a "T805").

Similarly, in mapping channels to links, the **PLACE .. AT** statement is used. Channels connecting processes residing on connected processors must be placed on the input link of one processor and the output link of the other. Processor links are identified by a link number.

Example

```
PLACE a.channel AT link0in :
```

where `link0in` is the defined hardware address of the input side of link 0.

A full example showing the various placement features is shown in figure 3.2.

Figure 3.3 shows a simple visualisation of the processors described by the configured program.

```
SC PROC buffer.1(CHAN OF INT in, out)
-- definition of single buffer process
  WHILE TRUE
    INT x :
    SEQ
      in ? x
      out ! x
  :

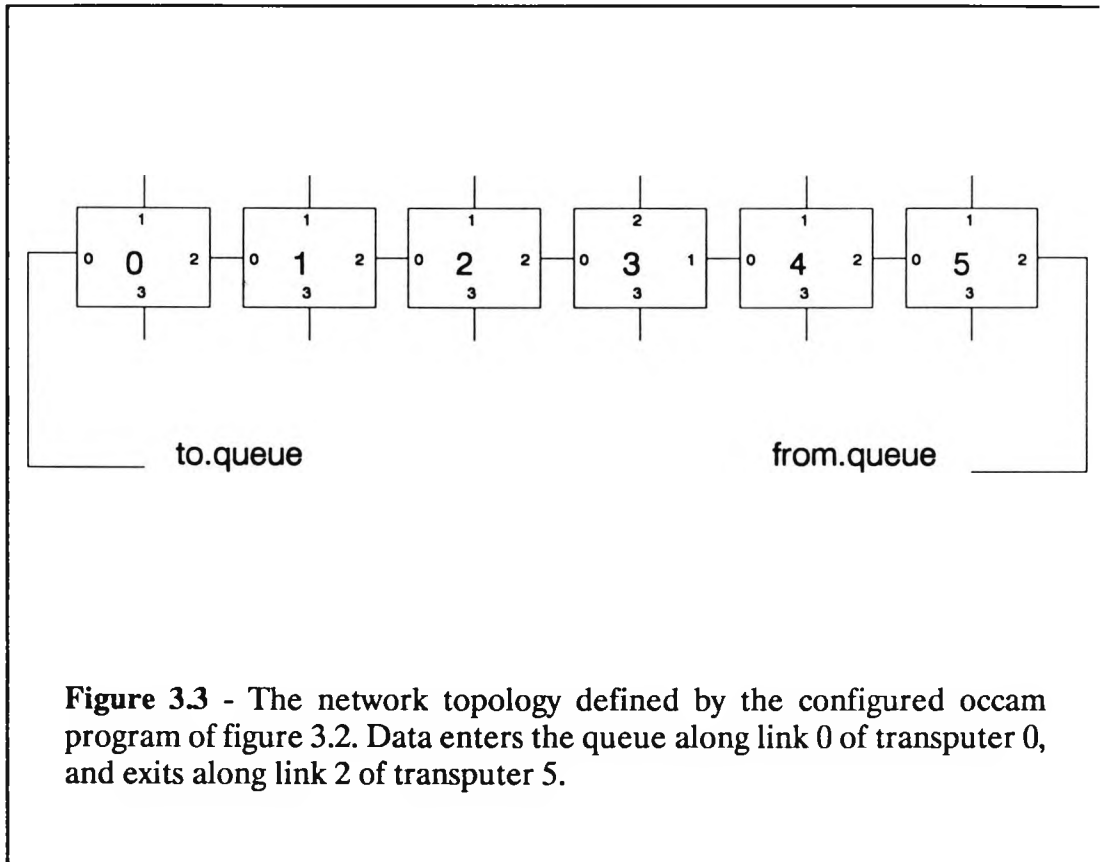
[5]CHAN OF INT link :
-- Main program
PLACED PAR
  PROCESSOR 0 T8
    PLACE to.queue AT link0in :
    PLACE link[0] AT link2out :
    buffer.1(to.queue, link[0])-- First buffer

  PLACED PAR i = 0 FOR 4
    PROCESSOR i+1 T8
      PLACE link[i] AT link0in :
      PLACE link[i+1] AT link2out :
      buffer.1(link[i], link[i+1]) -- buffers

  PLACED PAR
    PROCESSOR 5 T8
      PLACE link[4] AT link0in :
      PLACE from.queue AT link2out :
      buffer.1(link[4], from.queue) -- Final buffer
```

Figure 3.2 - A configured version of the program in figure 3.1. Link 0 is the input of each process, link 2 is the output. The constants link0in and link2out are assumed to be defined.

The procedure buffer.1 must be separately compilable, thus accessing no global channels or variables. Six processors are connected in a chain, with each processor running a separate process. In the TDS (Transputer Development System) a utility called the configurer is provided to check the correctness of the configuration and generate code for downloading into the processor array.



Configuration should not alter the logical behaviour of the program and is commonly performed after programs have been developed on a single processor. It should be noted that the programmer performing configuration requires knowledge of the underlying hardware architecture.

Clearly some help with configuration would be of considerable advantage. A graphical tool in which processes are interactively placed onto processors using a mouse or other pointing device is one possible solution.

3.4 Visualisations of Occam and of Transputer arrays

Graphical representations of Occam programs and of Transputer arrays have been widely used in the literature. The utility of diagrams in expressing Transputer related concepts is shown by authors' use of them in illustrating aspects of processor arrays and programs. This section discusses several different visualisations (graphical representations) of Occam programs and Transputer arrays and discusses each visualisation's merits and shortfalls for use in a visual programming environment.

3.4.1 Rack diagrams

"Rack diagrams" are commonly used to graphically represent the topology of Transputer networks. Transputers in a network are shown by squares. Links connecting Transputers are shown as lines or "wires" and are frequently annotated with relevant link numbers at each end. Rack diagrams are similar to diagrams found in papers on systolic arrays and in descriptions of other multiprocessor systems.

Figures 3.3 and 3.4 show such diagrams.

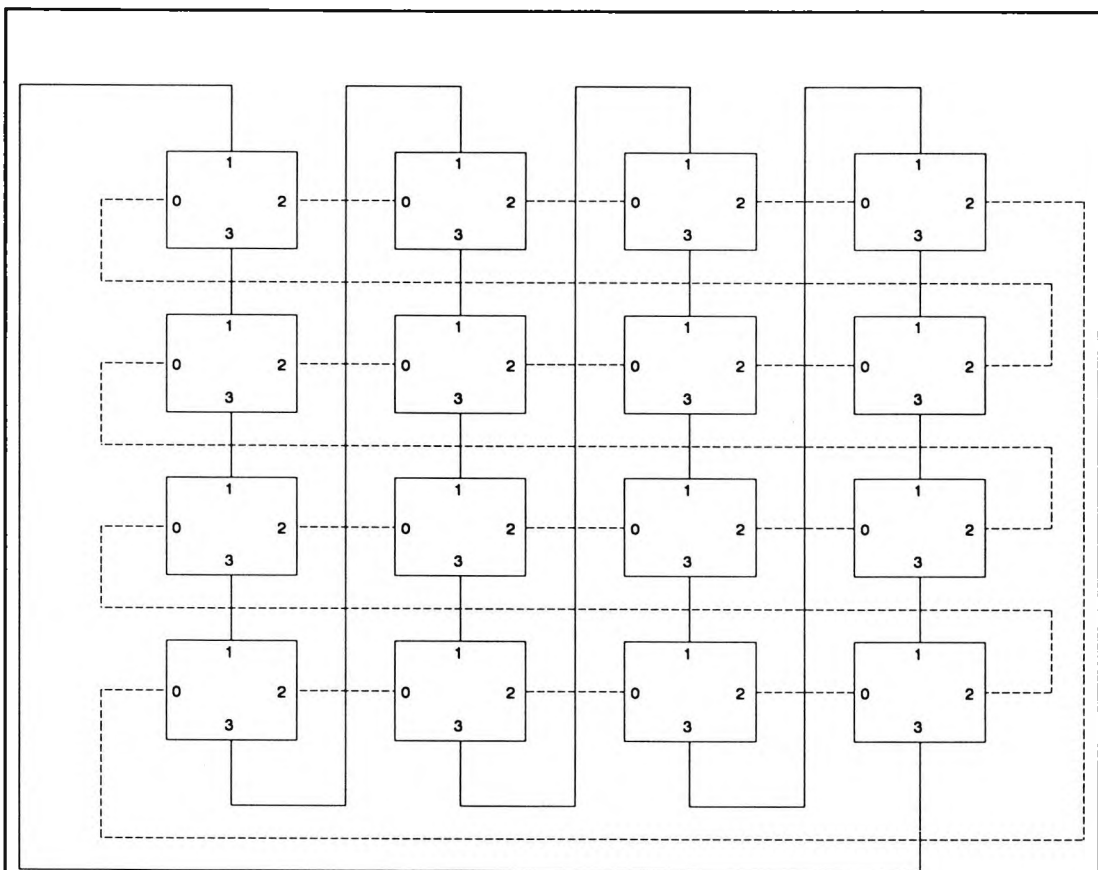


Figure 3.4 - A Rack Diagram showing sixteen transputers wired into a double ring topology. Different line styles are used to distinguish between the two rings. Double ring topologies are widely used in message routing systems.

The diagrams are extremely informative and a great improvement over a textual network map, as produced by the Inmos network worm (Inmos, 1989b). It is easy to imagine a visual programming system for the Transputer based on rack diagrams with a single, sequentially coded, Occam process for each Transputer. Connections between the four links on each processor could then be drawn to define the program's topology.

3.4.2 Bubble and arc diagrams

Bubble and arc diagrams are an informal diagramming method frequently used in the production, analysis and explanation of Occam programs. In the diagrams, Occam processes are usually represented by circles which are labelled with a suitable mnemonic (usually the process name). In some cases processes may be represented using square boxes, cloud shapes or "roundangles" (rectangles with rounded corners). Channels connecting processes are drawn as lines and are sometimes labelled with the channel's name. The diagrams are frequently used in papers on Occam programming. The diagrams bear a striking similarity to diagrams used in dataflow design methodologies (Gane and Sarson, 1979), and this is probably their origin. Figure 3.5 shows an example bubble and arc diagram. Sometimes bubble and arc diagrams are combined with rack diagrams to show the physical allocation of processes to processors.

A CASE tool for the design of deadlock free Occam programs based on the diagrams has already been discussed in the review section (Crowe, Hasson and Strain-Clarke, 1989). The tool is fully hierarchical. Bubbles representing processes may contain functional specifications in the form of a further diagram or as a textual definition written in a CSP derivative.

Another related system, Gecko (Stephenson and Boudillet, 1988), uses bubble and arc diagrams for the representation of Occam processes. The diagrams are laid out according to a "centre of interest" view. A process chosen for visualisation is placed in the centre of the screen. Connected processes are placed radially around the central process. In turn, other processes are laid out radially around the second set of processes and so on. The size of the coloured circles representing processes are scaled progressively smaller as the distance (in terms of communication "hops") increases. Gecko received its name from this display mechanism, which results in displays looking like geckos' feet.

Also relevant to the diagrams are many visual programming systems based on mixed textual and graphical paradigms, including IPigs (Pong, 1986), Conic (Kramer, Magee and Ng, 1989), STILE (Stovsky and Weide, 1987) and Poker (Snyder, 1984), all of which have already been discussed in chapter two. These systems all use bubbles or boxes for the representation of concurrently executing processes, while inter-process communication is shown using a variety of line styles. Textual specification is used for the expression of node functionality. All of the systems apart from Poker are hierarchical and allow the functionality of nodes to be expressed using further, lower level, diagrams.

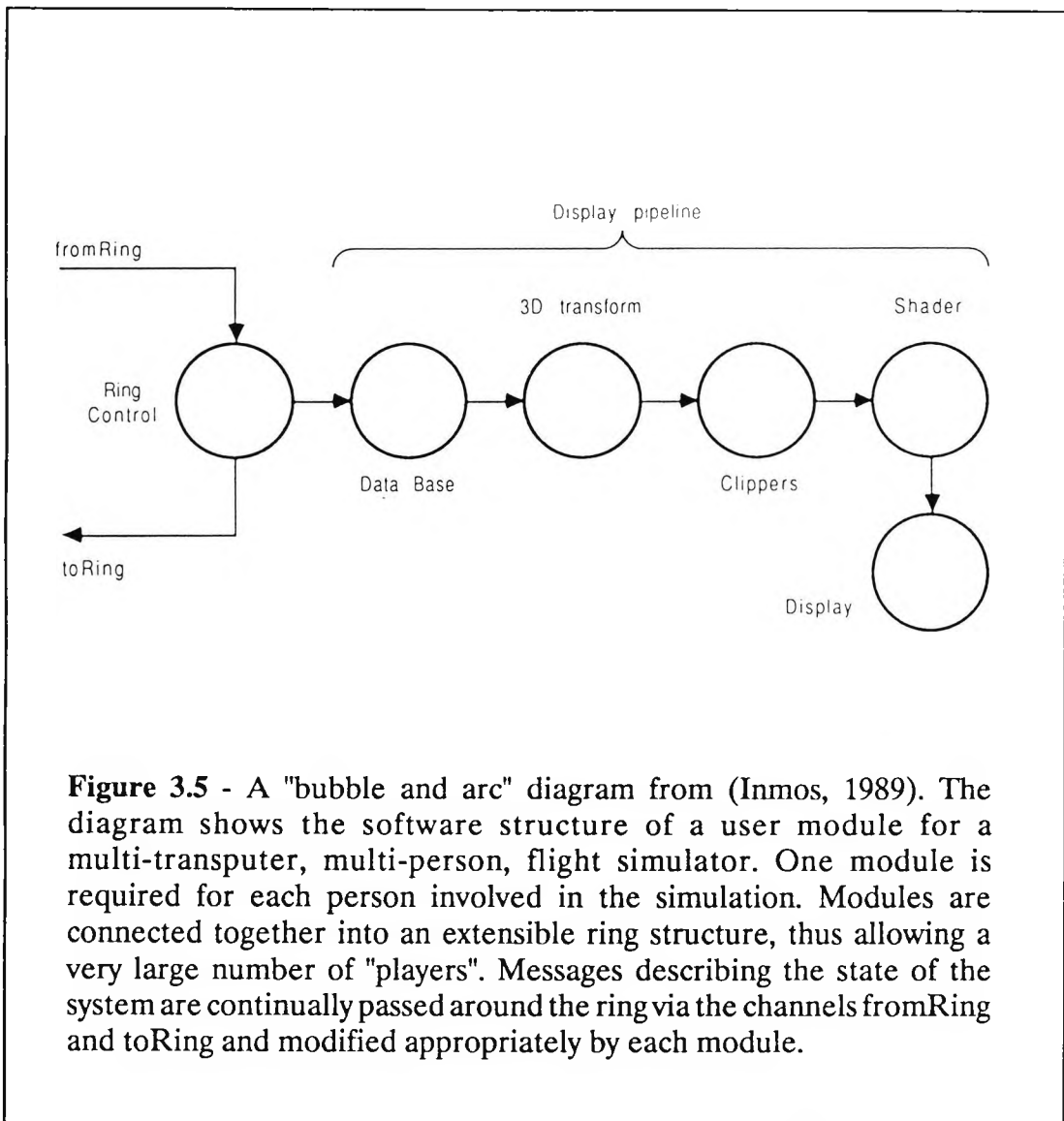


Figure 3.5 - A "bubble and arc" diagram from (Inmos, 1989). The diagram shows the software structure of a user module for a multi-transputer, multi-person, flight simulator. One module is required for each person involved in the simulation. Modules are connected together into an extensible ring structure, thus allowing a very large number of "players". Messages describing the state of the system are continually passed around the ring via the channels fromRing and toRing and modified appropriately by each module.

The usefulness of hierarchy in programming environments cannot be disputed. Amongst the many virtues it has for Occam style programming is that it is useful for aiding the mental organisation of processes into functional groupings. The usefulness of hierarchical organisation has been recognised for Occam programming in the folding editor of Inmos' Transputer Development System (TDS). The editor allows textual documents (usually Occam programs) to be created in a hierarchical style which reflects the structure of the program under development: "just as a sheet of paper may be folded so that portions of the sheet are hidden from view, the folding editor provides the ability to hide blocks of lines in a document. A fold contains a block of lines which may be displayed in two ways: open, in which case the lines of the fold are displayed between the two marker lines (called creases), or closed, in which case the lines are replaced by a single marker line called a fold line" (Inmos, 1988).

Kramer, Magee and Ng (1989) have termed the style of programming exemplified by systems like IPigs, Conic, STILE and Poker "configuration programming",

claiming that it not only provides a conveniently abstract form in which to comprehend programs but that it is particularly appropriate for distributed processing, where graphical software components reside on different machines. The graphical part of the program specification can then describe both the structure of the required system and the allocation of components to machines.

Bubble and arc diagrams (or similar) are ideal for the visualisation of high levels of abstraction in an Occam program's structure. Typically such high levels of abstraction consist of multiple parallel processes communicating with each other via channels. Clear visualisation of these structures does not require control flow be shown as all processes execute in parallel. From a macroscopic viewpoint the flow of control within the processes is determined by data flowing along the inter-process communications paths.

As the diagrams contain no control flow elements they are not suitable for the representation of low level Occam program detail. Low level Occam program detail frequently uses imperative control structures which, containing significant aspects of control flow, cannot be well represented using bubble and arc diagrams. Good examples of such structures are the alternative and conditional processes.

3.4.3 Control flow diagrams

Control flow diagrams have also been used for the representation of Occam programs, though not as widely as have bubble and arc diagrams. Control flow diagrams of Occam programs usually take the form of parallel flow charts. They are often used in the teaching of Occam programming and serve to illustrate constructs formed by parallel processes, multi-way conditionals, etc. They are very good at illustrating the relatively complex control flow structures occurring at low levels in programs. However, they are not so satisfactory for the illustration of higher level overviews of programs. This is for two reasons. Firstly, control flow diagrams are not particularly compact and thus do not provide good overviews. This can to some extent be alleviated by the use of hierarchical versions of the diagrams. Secondly, the diagrams do not provide the potential for the expression of structures involving inter-process communication. As high level views of Occam programs nearly always involve inter-processes communication, such diagrams cannot really be considered useful for programming at a high level of abstraction.

It is possible to overlay communication onto such diagrams. In (Mourlin and Cournarie, 1989) this approach has been taken to aid the visualisation of Occam programs. Parallel flowcharts are composed of Occam program fragments enclosed in boxes. By selecting various options, lines may be drawn between communicating primitive processes. Other options are included to "fold up" multiple processes into more compact units to aid analysis. Unfortunately, lines showing communication may only be drawn between unfolded primitive processes. The system therefore restricts the ability to obtain overviews of communication. Nonetheless, it does demonstrate that a combination of communication diagrams and control flow diagrams is a promising direction, at least for teaching. Figure 3.6 shows a diagram from the system.

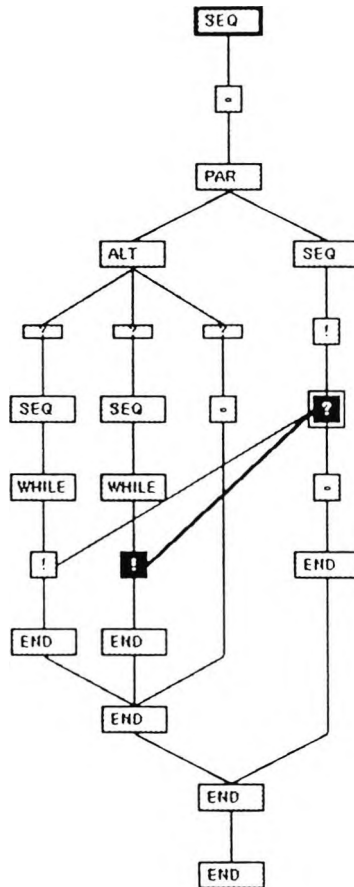


Figure 3.6 - A control flow diagram with overlaid inter-process communication information from (Mourlin and Cournarie, 1989). Boxes in the diagram represent occam processes. Communication is shown by the lines connecting the '!' and '?' symbols.

GRAIL (Stepney, 1987) is a system which uses a mixed control flow and channel visualisation paradigm. Unlike the previous system, GRAIL does not use "wires" for the representation of control flow connections between modules. Instead the flow of control is indicated by vertical and horizontal juxtaposition of "blocks" of program text in a manner similar to Nasí-Shneiderman diagrams. Channels are overlaid on top of the diagrams. Again, GRAIL has demonstrated that mixed approaches are useful.

3.5 Features of a visualisation of Occam

This section identifies general features for a visualisation of Occam, based on an analysis of the visualisations discussed in the previous section. Some general features for visualisations are presented and discussed with references made to how they might aid the development of Occam programs.

3.5.1 Visual representation of processes

All the visualisations of Occam discussed above use visual symbols (boxes or bubbles) for the representation of processes. Frequently boxes are labelled with process names. Many sequential visual programming systems have used icons for the visual representation of (non-concurrent) processes, and it is likely that similar techniques might be useful in concurrent programming.

3.5.2 Mixed representation of inter-process communication and of control flow.

Any visualisation of Occam style programs must include some form of graphical representation of inter-process communication. Inter-process communication is one of the building blocks of Occam. Bubble and arc diagrams have been extensively used for high level overviews of Occam programs and it is easy to postulate that the use of similar diagrams in visual program development environments is desirable.

Graphical representations which include control flow are able to aid the programmer in conceptualising the actions of the control structure of programs. Such a representation is therefore particularly well suited to the introduction of novice programmers to concurrent programming, and allows visual representation to extend to a lower level of abstraction that would usually be possible using bubble and arc diagrams.

The two methods may be considered as complementing each other, so a natural approach for an Occam style visual programming system is to combine the two into a single unified model. A system based on such a model should permit information hiding so that the complexity of the display may be reduced. For example, control flow information need not be displayed in situations where all processes are in parallel or the user has a clear mental model of the processes involved in the flow of control. A combined visualisation also allows the visual expression of control flow structures which contain elements of inter-process communication, for example alternative structures.

3.5.3 Use of graphics for high level overviews and text for low level detail

Graphical representations are well suited for showing the structural aspects of programs. Text on the other hand is better suited to the expression of abstract algorithms and low level program detail. Many systems have reported positive benefits from a mixed textual and graphical paradigm. Any system should leave the decisions on where to leave off visual programming and begin textual programming to the user.

3.5.4 Visual hierarchy

Hierarchy, as discussed earlier, can be of great facility. It aids the mental processes of software design by encouraging the functional grouping of processes, and has complexity management benefits. Hierarchical text editing systems are already well used in the development of Occam programs, so the expansion of the "folding editor" to a visual environment seems natural. Hierarchy encourages the development of small well founded program modules and hence supports code reuse. Finally, hierarchy is of considerable use in overcoming the restrictions placed on graphical programs by limited screen size and/or resolution.

3.6 The GILT language

The GILT visual programming language was designed with the features of the previous section in mind. Before an overview of the specific features of the language, some general design principles are analysed.

3.6.1 General design principles

A few obvious principles apply to the design of any prototypical language - it should be modular, extensible and modifiable. Within the bounds of practicality these principles were adhered to during the design of GILT. They proved to be very sound principles which allowed features to be introduced into the language as development progressed.

The major design decision taken was to combine the control flow and bubble and arc diagram approaches. The reasons for this decision have already been discussed, but it may be regarded as having the greatest effect on the language's syntax and semantics. Allowing control flow in GILT required the development of suitable visualisations of Occam control flow structures and various components associated with them.

Another fundamental decision was to allow the use of icons for the graphical representation of processes in the system. Simpler approaches might have used symbols composed of lines, or even labelled boxes. These were rejected in favour of the iconic approach as the use of icons gave greater flexibility by allowing the representation of a process to be customised by the user to show the process' function. The use of icons for the representation of sequential processes and applications has already been thoroughly investigated and proven useful (Shu, 1988; Yoshimoto, Monden, Hirakawa, Tanaka and Ichikawa, 1986; Glinert and Tanimoto, 1984; Lodding, 1983), so it seemed a natural step to allow their use for the representation of concurrent processes. The use of icons transpired to have little effect on the syntax or semantics of the language but had a great effect on the editing system. Icons may be regarded as decoration to the central principles of the language, or even as implementation dependent features. Nonetheless, they facilitate usage of the language.

For simplicity, the language was designed to include visualisations for a representative selection of Occam's programming structures only. For example, features from each class of Occam's control structures are included in the language - sequential execution of processes, parallel execution of processes, conditional execution of processes, iterative processes execution and indeterminate choice. Similarly, inter-process communication at the graphical level supports the use of simple integer protocols only. In all cases the simplest possible approach which was considered to have representative properties was taken. This approach has proven to be a good one as in hindsight it seems highly unlikely that a more complicated language could have been implemented within the time available for the project. Expansion of the system to fully implement all of the structures allowed in Occam is discussed in the final chapter of the thesis, but it should be noted that such expansion is not regarded as a complex matter, just a time consuming one.

The use of hierarchy was a feature which was identified as a promising property for any concurrent visual programming system. The most natural way of introducing hierarchy into any diagrammatic system is to allow diagrams to define the functionality of modules, which may then be used in further diagrams. This approach has been commonly used in many other visual programming systems as well as ECAD systems. Hierarchy was introduced in GILT by allowing a diagram to define the functionality of a "Process Icon", in turn requiring diagrammatic components for the definition of the external interface of a diagram together with representative counterparts at higher levels of abstraction. Similar approaches have been adopted in many of the aforementioned ECAD systems which use "pads" to define the external inputs and outputs of a circuit which is to be encapsulated into a diagram.

As an implementation and language specification aid, symbols in the language were built from the largest possible total number of common parts. In particular, icons showing the external connections of symbols are reused over and over again. This simplified the syntax of the language, and reduced the implementation time for the language's editor by allowing interaction with many different symbols to be described in a modular fashion.

3.6.2 An overview of the GILT system

Development of the language and a syntax capable of defining it were carried out largely simultaneously. Hence it is difficult to understand the syntactic representation of the language without some understanding of the language and vice-versa. Therefore, a brief synopsis of the entire system is given here as a basis for the discussion on syntactic representations suitable for formalisation of GILT diagrams in chapter four. A fuller description of the language is then given in chapter five.

The GILT language is based on a combined control flow and channel based visualisation, but uses iconic representation (Lodding, 1983) for language components. No previous visual programming system has used such a combined control flow and channel visualisation.

GILT programs are built at a workstation by interactive construction of hierarchical diagrams. The diagrams may be considered as graphs with labelled nodes and edges, which is how they are represented in the syntax developed for GILT. Nodes in the graphs are the basic components from which programs are built - Occam style processes, tags indicating the start and end of constructs, comments, variable definitions, inter-process communications facilities and other simple elements. Edges (arcs) between the nodes show the flow of control and inter-process communication. Nodes are visually represented by icons, some of which have textual labels, while edges are shown using two different line styles. The style of a line is dependent on the function of the edge which it represents. Processes may have graphical sub-processes or, at the lowest levels of abstraction, may be directly expressed in Occam. The graph model used allows the construction of programs consisting of small, potentially provable Occam processes connected together in a consistent and visual way.

GILT is designed to support a style of programming in which a user initially sketches a design of channel connected processes similar to bubble and arc diagrams, but with added control flow. Top level diagrams usually have all their processes executing in parallel, which is expressed by branching control flow to pass through all the processes. In such situations viewing control flow may be confusing. GILT therefore allows users to hide control flow from sight, so that graphs may be viewed using a pure communicating processes model. This facility is particularly useful when all the processes at a particular level of abstraction are in parallel, for example at the aforementioned higher levels of abstraction, or in the programming of a process array algorithm.

The functionality of the individual processes may then be expressed in further, lower level, GILT graphs until the user is satisfied that a sufficiently small level of granularity has been reached. Control flow constructs specifying the execution order of processes may be added as required to express the behaviour of processes which are not driven by the arrival of data along channels. Simple Occam code is then written for the lowest level processes. Graphical primitives for Occam operations like assignment, input and output are not included. These types of lower level operations are expressed at the textual programming level.

This "top down" approach is not equivalent to pure functional design, but produces hierarchical sets of functionally related processes which are expressed in a graphical sense by the use of a visual notation.

Alternatively, programs can be formed by wiring together pre-defined processes with system defined icons to form constructs. Iconic process labels are supported so that a process may be given a visual label appropriate to its function.

This approach is similar to the conventional engineering practice of bottom up design and has advantages for code reuse.

Realistically, both bottom up and top-down design methods may be used in practical program development. The inclusion of control flow into the graph model allows visual programming to proceed to a lower level of abstraction than is possible with the use of bubble and arc style diagramming notations.

GILT's icons do not provide functionality - they are not, for example, arguments to pre-defined processes as icons have been in some previous systems. Rather they provide a visual description of the functionality of the node which they represent. The pictographic representations used in GILT do not, however, replace textual descriptions for the complete representation of abstract concepts - GILT is based on a mixed textual and graphical model with text and graphics complementing each other. Textual process names, comments and variable definitions (just some examples) are as important in a program as are the program's visual aspects.

Users interact with the GILT editor by means of a mouse, menu and button based system. Icons representing processes (Process Icons) are drawn in a special icon editing area, then dragged to an appropriate position on the screen. Process Icons may be "entered" to reveal detail within them. Entering a Process Icon is equivalent to going down one level in the hierarchy to a lower level of abstraction. Text for the lowest level Process Icons is entered from the keyboard into pop-up windows. Other icons representing inter-process communication facilities, parts of control flow structures, etc. do not possess a hierarchical structure. These icons are simply placed in appropriate positions on the screen and connected to the Process Icons via "Control Flow Links" and "Channel Links" to form program structures. The Control Flow Links and the Channel Links are also defined with the mouse.

3.6.3 GILT diagrams

GILT diagrams are built from a small number of basic symbols called "functional icons" which may be connected into constructs using the Control Flow Links and Channel Links mentioned above. Constructs may in turn be connected together to form larger constructs.

Of the functional icons ("Process Icons", "Guard Icons", "Comments", "Variable Declaration Icons", "Passed Variable Icons", "Channel Stub Icons", "Control Flow Stub Icons", "Control Fork Join Icons", "Channel Connector Icons" and "Conditional Icons") the most fundamental is the Process Icon. Most of the symbols are composed of smaller sub-symbols to comply with the earlier aim of reducing the system complexity to a minimum. In most cases a symbol is composed of a unique iconic label, which visually represents the function of the icon, and a number of "ports", which provide connection points for Control Flow Links or Channel Links. In some cases editable areas for the entry of textual parameters are also provided.

GILT's icons may be regarded as having an hierarchical structure, as they are formed from a number of separate, nested, diagrammatic units. This hierarchical structure is described in more detail in the following chapters.

Process Icons are analogous to Occam processes. Each "Process Icon definition" contains either a few lines of Occam code or a "definition diagram". The appearance of a Process Icon's central image or raster and its name may be altered by the programmer to give an indication of the icon's functionality. A "Control Flow Input Port" and a "Control Flow Output Port" for the connection of "Control Flow Links" are positioned on the left and right hand side of instances of Process Icons placed into diagrams. As many "Channel Input Ports" and "Channel Output Ports" as are required for inter-process communication appear at the top and bottom of the "Process Icon instance". Figure 3.7 shows an example Process Icon instance.

Process Icon instances may be combined into control flow structures by connecting their Control Flow Ports to Control Flow Ports on other functional icons or to Control Flow Stubs (such connections are made via Control Flow Links). The other functional icons, the type of which is dependent on the structure being created, are either further Process Icon instances or system defined icons representing concepts like the branching of control flow or conditional control flow switching.

Similarly, inter-process communication structures are built up by the connection of Channel Ports and Stubs together via Channel Connector Icons and Channel Links. The Channel Connector Icons provide the ability to fork and join Channel Links and aid the implementation of some restrictions on the inter-process communication structures which may be created.

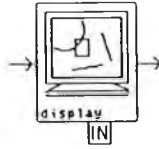


Figure 3.7 - A GILT Process Icon representing a data display process. To the left of the icon is a Control Flow Input Port which provides a connection point for incoming Control Flow Links. A similar connection point, a Control Flow Output Port, is positioned to the right of the icon. At the bottom is a Channel Input Port, which provides similar facilities to the Control Flow Ports, but for Channel Links.

External connections for control flow are provided by the use of Control Flow Input Stubs and Control Flow Output Stubs. Each diagram must contain only one Control Flow Input Stub and only one Control Flow Output Stub. The Control Flow Input and Output Stubs define the entry and exit points of the flow of control respectively. When an instance of a Process Icon defined by a diagram (a "graphical Process Icon") is used in a further diagram, the Stubs appear as the Control Flow Ports at each side of the icon. Control flow enters Process Icons via Control Flow Input Ports, and exits via Control Flow Output Ports. Ports give a high level visual representation of stubs which are at a lower level of abstraction.

A similar scheme is used for external channel connections, whereby Channel Input and Output Stubs correspond to the earlier Channel Input and Output Ports. Channel Links may be connected to Channel Input and Output Stubs just as they may be connected to the Channel Input and Output Ports of Process Icons.

Figure 3.8 shows a definition diagram for the Process Icon of figure 3.7.

The system of stubs and ports is similar to the schemes used in ECAD design systems for the hierarchical encapsulation of circuits into functional blocks. Stubs correspond to the external connections defined for a functional module in such a system, while ports correspond to connection points for the symbol representing the module.

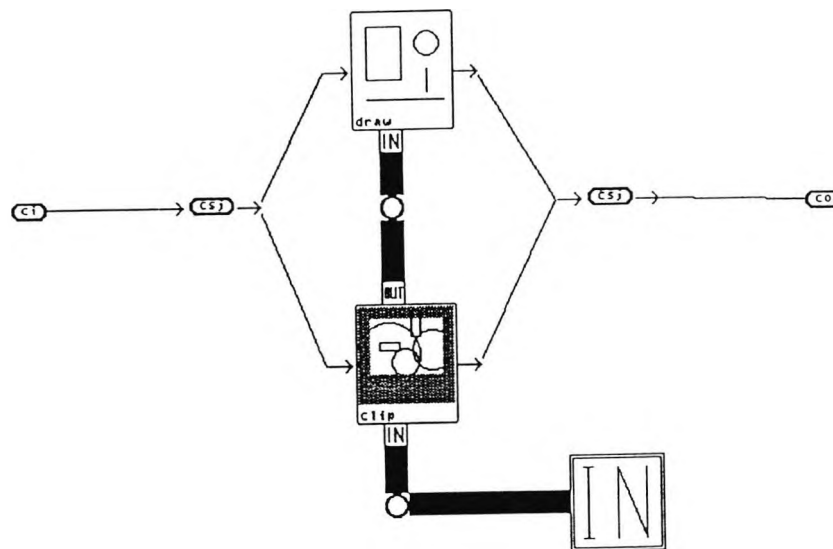


Figure 3.8 - A GILT definition diagram defining the functionality of the Process Icon of figure 3.7. Two communicating Process Icon instances, representing a clipping process and a drawing process, in a parallel construct are shown. Control flow (shown by the thin line style) enters the diagram at the Control Flow Input Stub (leftmost) and is divided to pass through the two Process Icon instances by a Control Split Join Icon. The control flow passes through the two Process Icons and is combined by another Control Fork Join Icon before passing out via a Control Flow Output Stub. Data may enter the system via the Channel Input Stub (bottom) which is connected to the Channel Input Stub of the 'clip' Process Icon instance via a channel link (thick line style). Processed co-ordinates are emitted on the process's Channel Output Port and enter the 'draw' Process Icon instance via its Channel Input Port. Channel Connectors (small circles) are used to connect Channel Links between Channel Ports and Channel Stubs, or between two Channel Ports. The Channel Input Stub defines the Channel Input Port shown on the icon of figure 3.7.

GILT's control flow structures are similar to those found in conventional flow charts and control flow graphs. Such representations have been widely disparaged as a design formalism, but the approach taken here is that hierarchical control flow graphs are well suited to the representation of imperative control flow structures in the implementation phase, at least for novice programmers. One difference between GILT diagrams and representations like control flow graphs and flow charts (apart from the obvious inter-process communication included in GILT diagrams) is that GILT's control flow structures use diagrammatic components to explicitly fork and join control flow links. Figure 3.8 shows an example of the use of these components, known as Control Fork Join Icons, in a very simple parallel construct.

Formal descriptions of visual languages

4.0 Introduction

A complete definition of any computer language is always required.

This chapter examines approaches to the development of formal descriptions suitable for the specification of visual languages. No great body of knowledge exists regarding the definition of the syntax and semantics of visual languages, but this chapter shows how textual language specification methods can be merged with current methods for visual system demarcation to provide grammatical and semantic specifications for visual languages similar to GILT. A notation for the representation of GILT diagrams is built up progressively through the chapter to provide a basis for a full definition of the language in chapter five.

4.1 Language, communication, and computer languages

Languages have been associated with communication since the beginnings of time. The study of languages is a complex arena consuming a vast research effort, with particular effort devoted by the computing community towards the understanding of "natural" languages and the problems engendered by "context". Yet, despite this effort, computer languages are some of the simplest of the diverse languages evolved by humankind. Their simple grammars and dictionaries allow them to be mechanically processed and thus facilitate the transfer of information between human and computer, which is surely our purpose here. Traditionally, computer languages are textually based. This can be regarded as due to the historical influence of limiting early technology. Just as did the printing process before it, computing has now developed methods for the practical reproduction (and storage) of images. It is the author's opinion that computer languages should be able to take advantage of these facilities to provide a new flexibility in the communication of ideas. However, the idea that purely visual languages can replace textual languages for the expression of abstract and concrete ideas is somewhat farfetched. Old adages like, "a picture is worth a thousand words" belie the fact that pictures and text complement each other and give fuel to proponents of views like, "nothing convincing, much less exciting, has emerged from such

efforts. I am persuaded that nothing will" (in reference to visual programming and program visualisation systems) (Brooks, 1987).

4.2 Specification of computer languages

The desirability of a formalised syntax and semantics for any computer language can hardly be disputed. Formal semantic and syntactic descriptions for textual languages have been in existence for many years, and it is almost impossible to consider the implementation of a textual computer language without a firm definition of its syntax. Semantic definition, although more problematical, is also highly desirable. Such definitions provide a solid basis for the development and management of software implementing the language.

4.3 Grammars and syntactic specification

Grammars are formal devices for the specification of potentially infinite languages in a bounded way. The syntax of a given computer language may therefore be defined using a grammar. Grammars generate language structures, which need not necessarily be textually based. This is done by successively rewriting a structure, consisting of a "start symbol", according to a finite set of rewriting rules or "productions". In traditional textual grammars, the structures consist solely of strings. In this chapter, structures of interest are graphs and textual sentences - hence the terms "graph grammar" and "text grammar" will be referred to where appropriate. In preparation for a definition of graph grammars, a revision of the definition of text grammars is first presented.

4.3.1 Text grammars and the specification of textual language syntax

A "text grammar" can be used to specify a textual language. Text grammars may be defined as follows, due to (Aho and Ullman, 1972), though it should be noted that there are many other equivalent definitions.

A set of symbols is termed a "vocabulary". The notation V^* , where V is a vocabulary, denotes the set of all strings composed of symbols from V , including the empty string. The "empty string", denoted ϵ , consists of no symbols. The notation V^+ denotes $V^* - \{\epsilon\}$. If α is a string, then $|\alpha|$ denotes the length of α .

A text grammar is a quadruple (V_t, V_n, S, P) where

V_t is a finite set of symbols called "terminals",

V_n is a finite set of symbols called "non-terminals" s.t. $V_t \cap V_n = \emptyset$,

S is a distinguished member of V_n , called the "start symbol", and

P is a finite set of pairs called "productions" s.t. each production (α, β) is written $\alpha \rightarrow \beta$ where the "left part" $\alpha \in V^*$ and the "right part" $\beta \in V^*$ where $V = V_t \cup V_n$.

In the following discussion, Latin capitals (A,B,...Z) are used to denote non-terminals, lower case Latin letters (a,b,...z) denote terminals, and lower case Greek letters ($\alpha, \beta, \dots, \omega$) denote strings.

If $(\alpha \rightarrow \beta)$ is a production and $\psi\alpha\rho$ is a string, then $\psi\alpha\rho \rightarrow \psi\beta\rho$ is an "immediate derivation". A "derivation" is a sequence of strings $\alpha_0, \alpha_1, \dots, \alpha_n$

where $n \neq 0$ such that

$$\alpha_0 \rightarrow \alpha_1, \alpha_1 \rightarrow \alpha_2, \dots, \alpha_{n-1} \rightarrow \alpha_n.$$

It is written $\alpha_0 \rightarrow^* \alpha_n$, a derivation ; or if $n \neq 1$ then $\alpha_0 \rightarrow^+ \alpha_n$, a "nontrivial derivation".

Any string derivable from the start symbol S, i.e. s.t $S \rightarrow^* \eta$ is called a "sentential form". Any sentential form consisting only of terminals is called a "sentence". The "language" $L(G)$ generated by a grammar G is the set of all valid sentences of the grammar;

$$L(G) = \{\eta \in V_t^* \mid S \rightarrow^+ \eta\}.$$

A text grammar is "ambiguous" if any strings in the language have two or more distinct derivations.

Text grammars may be classified into four types, ranging from type 0 to type 3 (DeRemer, 1976). The higher the classification, the more restrictions are placed on the productions in the grammar, and the easier the language is to parse mechanically.

Type 0 grammars, as defined by the unrestricted grammar above, generate "type 0 languages". The next three types place successively more restrictions on the form of the productions in the grammar.

Type 1 or context-sensitive grammars have productions of the form $\psi A \rho \rightarrow \psi \omega \rho$ where $A \in V_n$ and $\psi, \omega, \rho \in V^+$. A context-sensitive grammar generates a "type 1 language".

Type 2 or context free grammars have productions of the form $A \rightarrow \omega$ where $A \in V_n$ and $\omega \in V^*$. Sometimes ω is not allowed to be the empty string ϵ . A context free grammar generates a "type 2 language".

Type 3 or regular grammars are either right linear with productions of the form $A \rightarrow a$ or $A \rightarrow aB$, or left linear, with productions of the form $A \rightarrow a$ or $A \rightarrow Ba$, where $A \in V_n$, $B \in V_n$ and $a \in V_t$. A regular grammar defines a "type 3 language".

The classification above is generally known as the Chomsky hierarchy. Computer languages are generally of type 2 and the specification of their syntax may therefore be accomplished using a context free grammar. Backus-Naur Form (BNF) is arguably the most common notation for the description of context free grammars, and numerous examples of its use are to be found in texts on programming languages.

4.3.2 Graph grammars for the specification of visual language syntax

Most diagramming or visual programming "picture models" represent discrete, limited structures. These structures consist of a finite number of discrete objects interrelated by a finite number of relations. Graphs with discrete objects and edges map onto such diagramming methods. Thus a picture may represent a graph by visually depicting its nodes and edges. This mapping is not usually a one-to-one mapping of picture to graph. Rather it is frequently a many-to-one mapping with multiple sets of pictures representing single underlying graphs. Representational pictures may therefore contain a degree of information redundancy, for example in the domain of spatial information. Some spatial information may be relevant to the semantics of a given visual language but it usually has little or no relationship to the language's syntax.

Thus it is possible to describe the syntax of a visual or picture based language by the use of a graph grammar. The complexity of the graph grammar is obviously related to the complexity of the language it is required to describe. Although graph grammars are more complex than conventional grammars, some graph grammars can be viewed as generalisations of text grammars. Most of the terms used in the study of the text grammars discussed earlier still apply to such graph grammars.

4.3.2.1 Previous work on the syntactic descriptions of visual languages

Most visual programming languages have been based on highly informal syntactic definitions having little theoretical basis, and although more formal syntactic descriptions of visual languages have recently been gaining in popularity, the majority of visual languages are still heuristically based.

The syntax of many visual languages may be described using graph grammars. Like textual grammars, these grammars can exhibit interesting properties and can be specified in many different ways. Unlike textual languages there is no standard or even widely used method in existence for syntactic specification. Instead, a variety of different graph grammar based methods have been applied to the problem. Even the term graph grammar is a hazy one, referring as it does to a multitude of methods for the specification of sets of graphs or maps developed since the late 1960's. Graph grammars have been applied in pattern recognition, molecular modelling, VLSI layout schemes, data bases, lambda-calculus, and a host of other diverse activities.

The use of graph grammars in the field of visual languages has not been confined to the specification of the syntax of visual languages. For example, several papers have proposed the use of graph grammars both as a tool for syntactic definition of visual languages and as an aid in their implementation (e.g. Harada and Kunii, 1984; Gottler, 1989). In particular graphical syntax directed editors, which may be customised by the use of a graph grammar description of a visual language's syntax, have been proposed to cut down on the often lengthy business of developing an editor for a visual language (e.g. Gottler, 1989; Hekmatpour and Woodman, 1987). Relevant syntax directed editors have also been developed for diagramming methods (e.g. Inman, 1987; Dutton, 1986; Albizuri-Romero, 1984; and Szwillus, 1987). These tools are still in their infancy, and are unable to deal with visual languages of realistic complexity. Certainly none of these systems would be suitable for the implementation of a GILT-like visual programming language, mainly due to their lack of support for bitmapped icons and their lack of support for the multiple levels of abstraction required by GILT diagrams.

Another relevant work in the area is Lakin's paper on spatial parsing (Lakin, 1987), which is concerned with formalising human-computer diagram based interaction. Such interaction can be regarded as the foundation underlying visual programming, program visualisation and other schematically based systems.

Work has also been performed on the representation of textual programs using hierarchical graphs (Pratt, 1971; Yau and Grabow, 1981), which are discussed in detail in later sections of the thesis and will not be further mentioned here.

Harel (1988) has reported work on the theory of "Hi-Graphs". A Hi-Graph is a general kind of diagramming object formed by the replacement of nodes in hypergraphs with visual representations of sets. Hyperedges then express relationships between sets of objects. Harel does not offer methods for the specification of particular classes of Hi-Graphs, instead concentrating on the underlying theory of the diagrams. A graph-grammar like system could be developed for use with Hi-Graph theory, which offers a high level of generality. Such an approach is not taken here as most of the features offered by Hi-Graphs are not required for the visual representation of GILT or many other visual languages. The graphs generated by the simple graph grammars offered below as a syntactic formalism may however be regarded as equivalent to simple Hi-Graphs.

In an approach very different to the ones described above, the syntax of visual languages has been defined using textual formalisms. The major difficulty with such approaches is the specification of the mapping between the two dimensional graphical language being described and the one dimensional textual representation used to describe it. Another associated difficulty lies in finding a suitable grammar for the representation of the sets of the combinations of textual elements used. Gillett and Kimura (1986) have used such an approach for the Show and Tell language, while a less formal, though similar, system has been used for the syntax of Prograph (Cox and Mulligan, 1985). The lack of a formalised equivalence between the representations used and difficulties with suitable grammars for the textual specification makes such methods unattractive, at least for GILT's syntax.

4.3.2.2 Graph grammars

Graph grammars provide similar facilities for visual languages that text grammars provide for textual languages. As mentioned earlier there is a wide diversity of graph grammars in existence, with many methods for their specification. All graph grammars specify graph languages in a manner similar to the way in which text grammars specify text languages. Instead of working with strings the grammars work with graphs and produce languages of terminal graphs. Rather than specifying the replacement of strings of symbols with strings of symbols, productions in the grammars specify replacements of sub-graphs in host graphs with other sub-graphs. The various graph grammar specification methods differ from each other in the way in which the "embedding transformation" is specified. The embedding transformation defines the way in which the new sub-graph's edges are connected to the original graph. The task of reviewing the entire spectrum of approaches, contrasting the various methods and reviewing applications has already been accomplished and therefore will not be undertaken here. Ehrig, Nagl, Rozenberg and Rosenfeld (1986) includes tutorial introductions to the major methods but a few references to the major approaches are perhaps appropriate at this point. The most well known methods are the algebraic method (Ehrig, 1979), the NLC (Node Label Control) approach (Rosenberg, 1986) and the set-theoretic approach (Nagl, 1986a).

The approach developed in this chapter is a form of NLC graph grammar, but has directed edges which are not usually found in NLC grammars. The grammar is based on work by (Pratt, 1971), (Della-Vigna and Ghezzi, 1978) and is a form of context free graph grammar. The method was chosen because it offered sufficient generality to describe the constructs of the GILT language, exhibited properties of context freeness (and hence generated a grammar that is easy to parse), yet was relatively simple in both its description and theory. A context free grammar is required, at least in the representation of the control flow information in GILT, to allow the easy recognition of GILT's language constructs.

Before describing the grammar itself some underlying terms must be defined. In both the definition of the graph grammar and the preamble, attempts are made to draw parallels between the definitions of textual grammars and graph grammars, even to the extent of using the same symbols for like concepts.

A set of symbols is termed a "vocabulary". Assume that V_M and V_A are finite sets of distinct symbols. V_M and V_A are the sets of node labels and arc labels respectively.

Definition :

A "labelled graph" G over V_M and V_A is a triple (N, L, E) where N is a finite set of nodes.

$L: N \rightarrow V_M$ (L , the node labelling function, defines the label of each node).

$E \rightarrow (N \times V_A \times N)$ (E - the arc set, defines the arcs of G and their labels).

If $(n, a, m) \in E$, then an arc exists from node n to node m with label a .

If G is a graph, then N_G , L_G and E_G denote the node set, node label function and arc set of G respectively.

Definition :

If V_M and V_A are finite sets of distinct symbols then the vocabulary $V^*(V_M, V_A) = \{G \mid G \text{ is a graph over } V_M, V_A\}$. In shorthand, where V_M and V_A are assumed it is written V^* . In equivalence to the vocabulary in the earlier text grammar, V^* is the set of all graphs composed with nodes and arcs having labels from V_M and V_A , including the "empty graph".

The empty graph, denoted ϵ , has no nodes or edges. The notation V^+ denotes $V^* - \{\epsilon\}$.

A "graph grammar" is a quintuple (V_t, V_n, V_a, S, P) where :

V_t is a finite set of "terminal" node labels (the "terminals"),

V_n is a finite set of "non-terminal" node labels (the "non-terminals"),

V_a is a finite set of arc labels (the "arcs"),

S , the "start symbol", is a distinguished member of V_n ,

and P is a set of "productions" s.t. each production is a quadruple (G, H, I, O) and written $G \rightarrow H, I, O$. The "left part" $G \in V_N$. The "right part" is H, I, O . In general $H \in V^*$. In our case we do not allow H to be the empty graph ϵ and hence $H \in V^+$. I and O are distinguished nodes in H termed the "input" and "output nodes" (or "gluing points") respectively.

Productions are used to derive graphs with node labels in V_T starting from a "host graph" containing only the start symbol S . During the derivation, the nodes of the host graph with labels in V_N (e.g. an arbitrary node A labelled B , with $B \in V_N$) are replaced by the right part of some rule rewriting B , e.g. $B \rightarrow C, I, O$. Every arc originally entering (exiting) the node B becomes an arc entering I (exiting O). Thus I and O define the "embedding" of the right part (C) in the host graph. Thus the way in which the "embedding transformation" is specified in the grammar is extremely simple.

A graph grammar is "ambiguous" if the language that it generates contains a graph with two or more distinct derivations.

Although Chomsky's hierarchy clearly applies only to textual grammars it is easy to see that some of the notions of context freeness (type 2 grammar) can be transferred to graph grammars. A context free graph grammar, like the one above, has productions which replace only single non-terminal nodes with graphs. A context dependent graph grammar would have productions in which sub-graphs were replaced with other graphs. The embedding mechanism of such a grammar would have to be considerably more complex than that used above, which has only two gluing points (I and O) defining the connectivity of the substituted graph with the host graph. Obviously a completely general graph grammar with unrestrained substitution would correspond with Chomsky's type 0 grammar. Regular grammars pose more of a problem however. Any attempt to classify graph grammars into a hierarchy like Chomsky's would have to remove the notion of right and left linear grammar, as there is no "right" or "left" to most graph grammars, as they do not take account of the spatial positioning of nodes.

4.3.2.3 A simple visual language grammar

A good example to begin with is a grammar for a "box language". The box language is a trivial language - its input space consists of chains of simple boxes, one connected to the next. At the start and end of the chains of boxes are connected a start box and an end box. Each chain of boxes must have one start box, one end box, and at least one middle box. Clearly any chain of boxes in the language can be modelled as a simple directed acyclic graph with labelled nodes. The problem is to find a simple graph grammar description for the language which allows chains of boxes of all lengths to be developed, and hence legal chains of boxes to be mechanically parsed. In a sense, such a simple box grammar is very like a string grammar - boxes are equivalent to characters, but the connections between the boxes are modelled explicitly. Connections between characters in string grammars are not shown - they are implicit in the grammar. The grammar for such a simple language is a good place at which to introduce the terminology and methods described above in a practical way.

Terminal symbols

Terminal symbols in the grammar representing the different box types are shown as labelled nodes. Three exist - the start box, which we shall label [SB], the end box [EB] and the intermediate boxes [B]. All terminal symbols are enclosed by square brackets. In later, more complex, examples terminal symbols will be enclosed in rectangles.

$$V_T = \{ [SB], [EB], [B] \}$$

Non-terminal symbols and productions

Non-terminal symbols in the grammar are enclosed by round brackets. Later, more complex, examples have non-terminal symbols enclosed by rounded rectangles.

$$V_N = \{ (GRAPH), (BOXES) \}$$

The productions in the grammar may be written in a similar manner to BNF productions :

P = {

$$(\text{GRAPH}) ::= {}^I[\text{SB}] \rightarrow (\text{BOXES}) \rightarrow [\text{EB}]^O \quad (1)$$

$$(\text{BOXES}) ::= {}^I[\text{B}]^O \quad (2)$$

$$::= {}^I[\text{B}] \rightarrow (\text{BOXES})^O \quad (3)$$

}

The input and output nodes (or gluing points), as defined earlier, are denoted by "I" and "O". The " $\rightarrow$ " symbol indicates an arc (edge) between symbols. In later, more complex, examples arcs are shown by directed lines. The bracketed numbers are used to reference the productions in later examples.

Start symbol

The start symbol, V_N , which must be part of the set of non-terminals is (as expected) the symbol GRAPH.

S = (GRAPH)

Arc Labels

All arcs in the diagrams have the same label. Hence, for simplicity, no arc labels are applied. In other cases, different style lines may be used to avoid writing many labels.

Rewriting

The two productions above are enough to describe any legal box graph. Rewriting begins with a single (GRAPH) node and proceeds until no non-terminal nodes exist in the graph.

Example

Initially a graph consists of the start symbol :

(GRAPH)

Applying substitution (1) we obtain :

[SB] $\rightarrow$ (BOXES) $\rightarrow$ [EB]

As there were no connections to the initial symbol, the gluing points of productions (1-3) were redundant. They will however be required in future rewritings of the (BOXES) symbol, which has connected arcs.

We now have a choice of substitution to make for the non-terminal (BOXES). Selecting the (3) substitution yields :

[SB] -> [B] -> (BOXES) -> [EB]

Arcs entering the original (BOXES) symbol are connected to the node labelled [B], while those connected outwards from the original (BOXES) symbol are connected from the new (BOXES) symbol, as defined by the gluing points.

Clearly substitutions for (BOXES) could continue indefinitely. Instead we select (2) giving :

[SB] -> [B] -> [B] -> [EB]

which contains no non-terminal nodes and is clearly a legal graph. This notion of rewriting until no non-terminal nodes exists in the graph is fundamental to the method.

4.3.2.4 A graph grammar for a subset of GILT

A slightly more complex example of a visual language, which can be regarded as defining a subset of the GILT language, may be developed with simple sequential and simple parallel control flow constructs. The language is flowchart-like, is without conditionals or indeterminate choice of any kind but allows one parallel control flow construct. Graphs in the language are composed of networks of processes "wired" either sequentially or in parallel, like sequential and parallel constructs in GILT.

Constructs in the language, modelled by the non-terminal "CONSTRUCT" include a sequential construct (non-terminal "SEQ"), a parallel construct (non-terminal "PAR"), a "SKIP" symbol (terminal) and a single process ("PROCESS INSTANCE" terminal). The terminal "CONTROL SPLIT JOIN" is used to fork and combine control flow. As in Occam, sequential and parallel constructs may be nested as deeply as required. The start symbol is "PROCESS DEFINITION".

The grammar defines the syntax of a simple subset of GILT by representing GILT's icons with labelled nodes. Control flow links are represented by edges. The representation does not model the hierarchical structure of the icons, leaving them instead as simple non-hierarchical terminal symbols.

In Occam the notation of a set of processes (using { }) is an essential part of Occam's grammar (appendix one). It allows SEQ, ALT, and PAR constructs to be easily defined and simply written.

For example, the production

```
parallel ::= PAR
          {process}
```

defines the non-terminal "parallel" as being the terminal "PAR", followed by zero or more occurrences of "process" non-terminals.

A similar notation is required for the elegant representation of sets of processes wired between two common nodes such as those in a parallel construct. Infinite sets of sequential processes can easily be represented using recursive productions, as in the box grammar.

Therefore, Occam's set notation is extended to encompass graphical parallel sets of processes. In such a parallel set, a pair of horizontal braces delimits a construct that may be replicated vertically in the production. For example, figure 4.1(c) contains a production for the parallel construct in the simple language. The parallel construct can consist of one or more of the branches shown between the braces, provided that each branch is connected to the terminals labelled "CONTROL FORKJOIN".

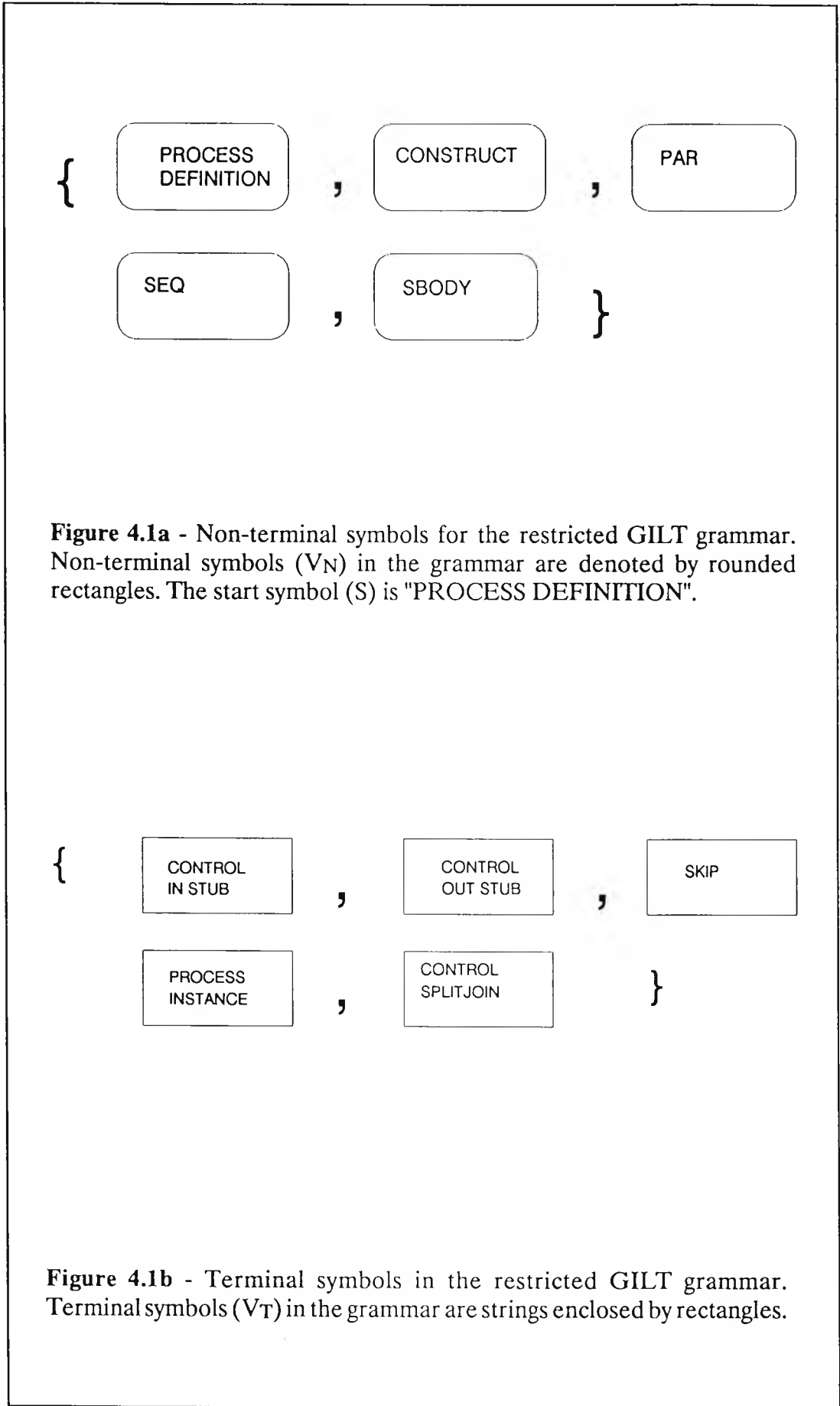
Figure 4.1 shows the grammar for the restricted subset of the GILT language, including non-terminal symbols (4.1a), terminal symbols (4.1b) and productions for the grammar (4.1c). Only one edge label exists, as before, and is shown by lines using a thin line style.

Figure 4.2 shows a simple example graph, containing sequential and parallel constructs. The parallel constructs are worthy of note, having been generated using productions expressed using the earlier set notation.

4.3.2.5 Hierarchy and communication

The grammar above can easily be extended to provide a complete description for non-hierarchical diagrams without a mixed inter-process communication and control flow visualisation. Such a grammar is capable of describing visual expressions for all common block structured control constructs, but the grammar cannot express mixed inter-process communication and control flow diagrams or multi-level (hierarchical) graphs.

Two extensions are required to allow the grammar to fully represent GILT diagrams. Firstly, some sort of hierarchical structure is needed so that processes may contain sub-processes to an arbitrary level of abstraction. Secondly, some way of expressing inter-process communication in the same diagram as control flow connection is required.



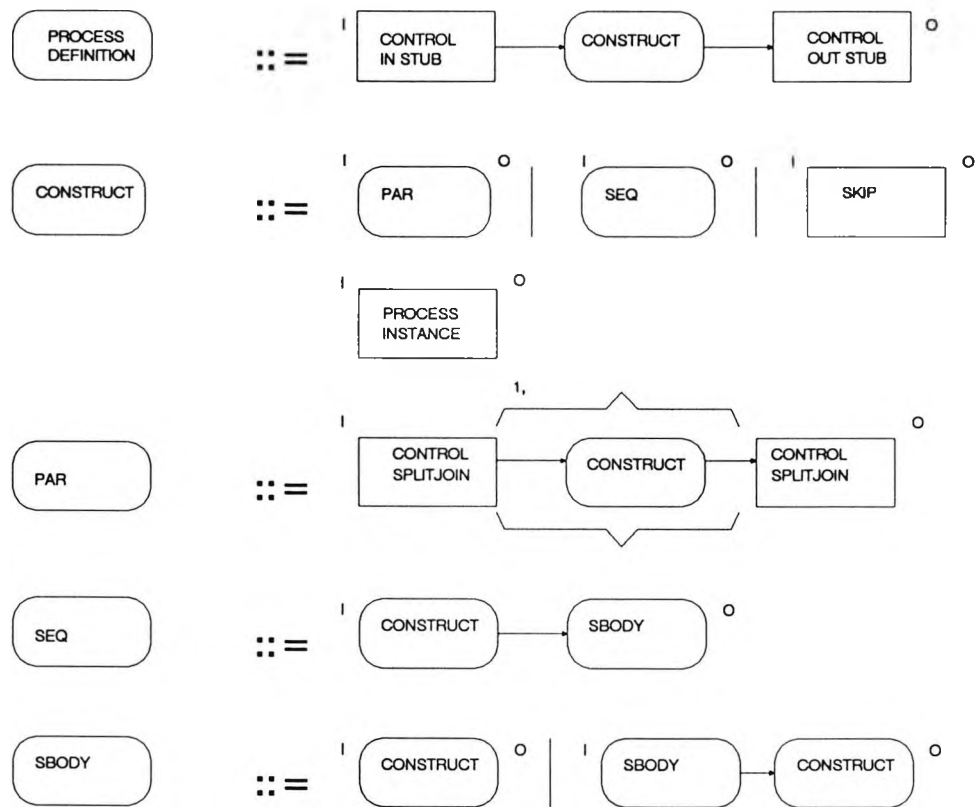


Figure 4.1c - Productions in the restricted GILT grammar. Productions (P) are written in a BNF like notation. The input and output glueing points are denoted by I and O. Only one edge label exists in the grammar. Edges are therefore assumed to all be labelled with the same label.

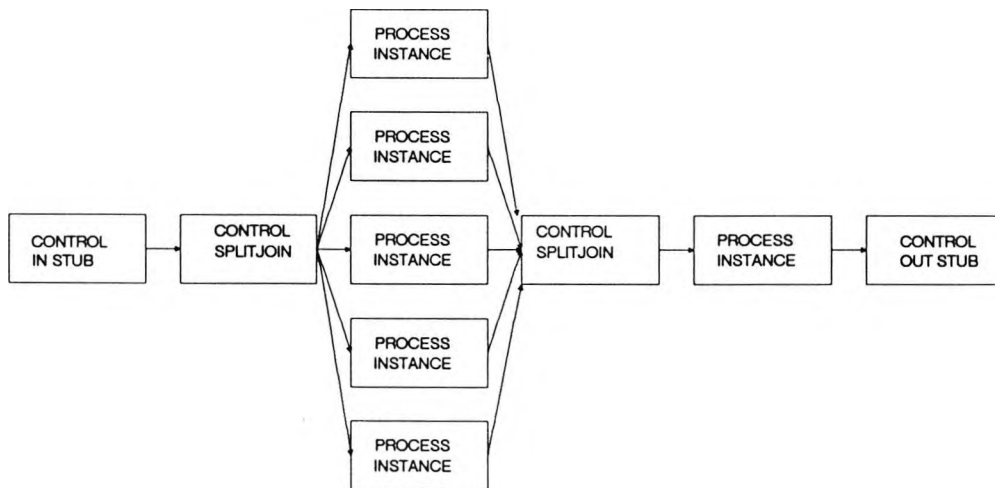


Figure 4.2 - An example graph generated using the grammar of figure 4.1. The graph was generated using rewriting as described for the box grammar.

4.3.2.5.1 Hierarchy

Hierarchy can be added to the definition of a graph by replacing the earlier node label function with a "node value function" which allows the value of a node to be a terminal symbol or a graph. Structures composed of hierarchies of graphs may be expressed using such a model, as follows: The top level in a hierarchy is a single graph. Each node in this graph has a value which is either a terminal symbol or a graph, which could be called a second level graph. Each second level graph in turn contains nodes whose values are again either terminals or third level graphs, and

so on. Ultimately, the lowest level in the hierarchy contains graphs which have only terminal values.

Grammar rules in such a modified system rewrite non-terminal nodes as graphs which may contain terminals and non-terminals, as before, or additionally, other graphs whose nodes in turn may contain terminals, non-terminals or graphs to any (finite) depth. As context free graph grammars can be thought of as generalisations of context free string grammars, even textual strings can be integrated into such a hierarchical graph model by modelling connections between characters in textual sentences explicitly, as in the earlier box language.

The definition of a labelled graph is therefore expanded to that for a "hierarchical valued graph" (henceforth, an "H-graph"), using similar notation to that used in the definition of the labelled graph.

4.3.2.5.1.1 H-graphs

In a hierarchical valued graph, the notion of a node label is replaced by that of a node value.

A hierarchical valued graph or H-graph over V_M, V_A is defined as follows:

The vocabulary of a level-0 H-graph $H_0^*(V_M, V_A) = V_M$.

A level-1 H-graph over V_M and V_A is a valued graph over V_M and V_A . The valued graph is as the labelled graph defined in section 4.3.2.2, except all references to node label are replaced by references to node values. The full definition for a valued graph is reproduced in appendix 2.

$H_1^*(V_M, V_A) = V^*(V_M, V_A)$, the set of all level-1 H-graphs.

A level-k H-graph ($k \geq 1$) over V_M, V_A is a graph over

$\bigcup_{i=0}^{k-1} H_i^*(V_M, V_A)$ providing that V_A

has at least one node value in $H_{k-1}^*(V_M, V_A)$.

In shorthand, $H_i^*(V_M, V_A)$ is written H_i^* .

$H_k^*(V_M, V_A) = \{X \mid X \text{ is a level-}k \text{ H-graph}\}$

$H^*(V_M, V_A) = \bigcup_{k=0}^{\infty} H_k^*$ is the set of all H-graphs over V_M, V_A , in shorthand written H^* .

The notation H^+ denotes $H^* - \{\epsilon\}$.

4.3.2.5.1.2 An H-graph grammar for a subset of GILT

Productions are allowed to rewrite non-terminals to graphs whose node values may be terminals, non-terminals, or further graphs. These further graphs may in turn contain terminals, non-terminals or graphs. The "depth" of the rewriting is restricted to be finite.

The definition of the earlier labelled graph grammar and the definition of a H-graph grammar are so similar that a complete reproduction of both is unnecessary here. Indeed, extension of the earlier labelled graph grammar definition to cover H-graphs requires only a few simple modifications, though for completeness a definition for the H-graph grammar is given in appendix 1. The modifications that need to be made to the earlier labelled graph grammar to expand it to an H-graph grammar are as follows: Firstly, V^* is replaced throughout by H^* . Secondly, the notion of a node label is replaced by that of a node value and finally, the input and output nodes of a production are constrained to be in the top level of the right side of a production.

4.3.2.5.1.3 The simple grammar with hierarchy.

The earlier simple grammar of figure 4.1 is expanded by replacing the "PROCESS INSTANCE" terminal symbol with a "PROCESS INSTANCE" non-terminal. A production rewrites this non-terminal to a node with a value that is a "PROCESS DEFINITION" non-terminal or to a terminal textual process. The "SKIP" symbol is also modelled in greater detail by a production which rewrites a "SKIP" non-terminal into a node with a value which is a graph consisting of a "CONTROL IN STUB" terminal connected to a "CONTROL OUT STUB" terminal. Thus hierarchical graphs of any required depth and complexity may be developed. The grammar is considerably expanded, but lacks any representation of inter-process communication. More control flow structures such as while loops, if..else structures, etc. may easily be added, however.

Nodes with values which are graphs are shown in the rectangular terminal style with their value graph enclosed in the confines of the rectangle.

Figure 4.3 shows the expanded grammar, with figure 4.4 showing an example hierarchical graph.

4.3.2.5.2 Communication

Inter-process communication must also be included in a definition of GILT's syntax. Extensions to the previous graph model are required because a context free graph grammar cannot express all possible inter-process communication patterns even though it is possible to include more syntactic information in a graph grammar

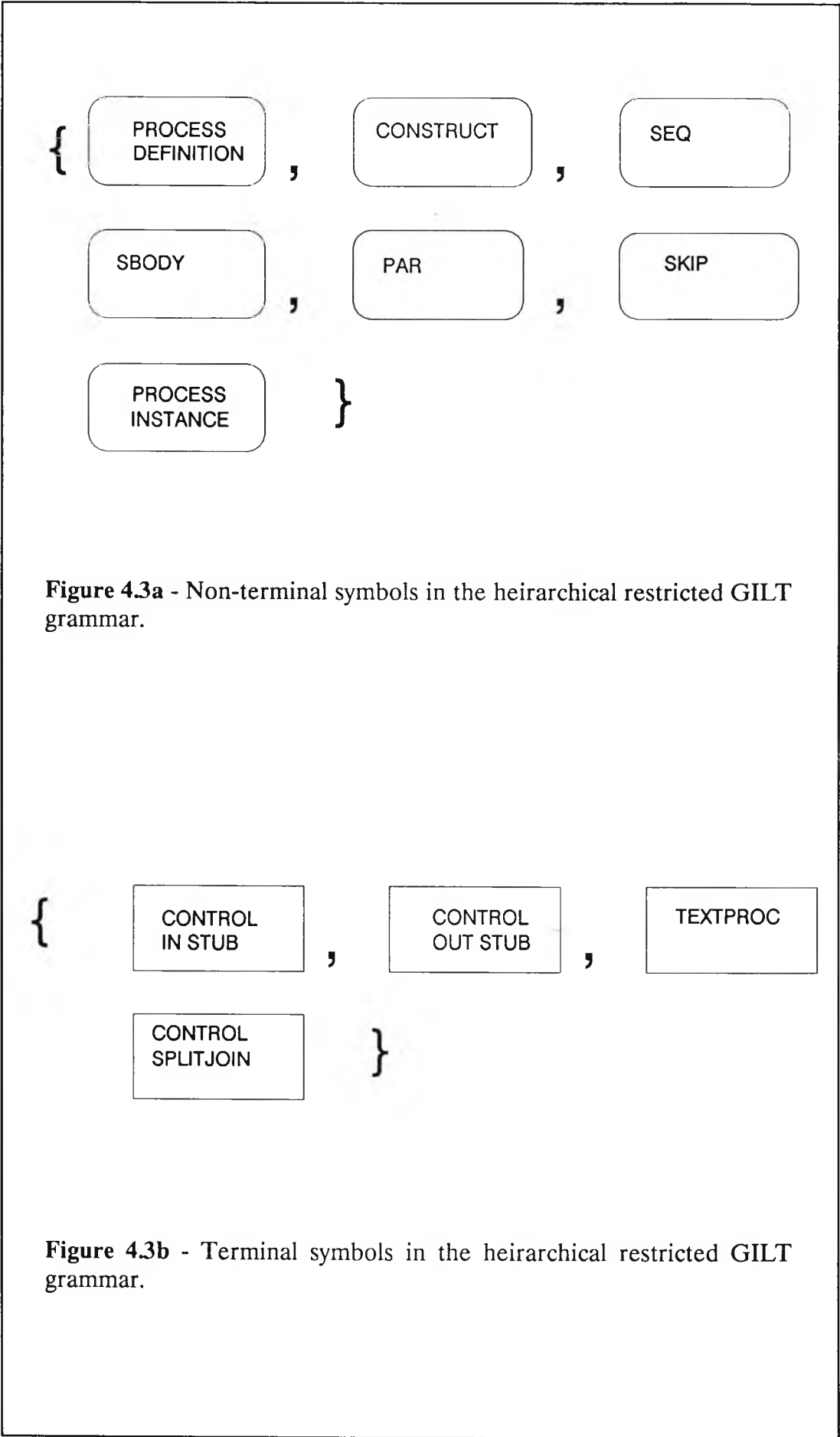


Figure 4.3a - Non-terminal symbols in the heirarchical restricted GILT grammar.

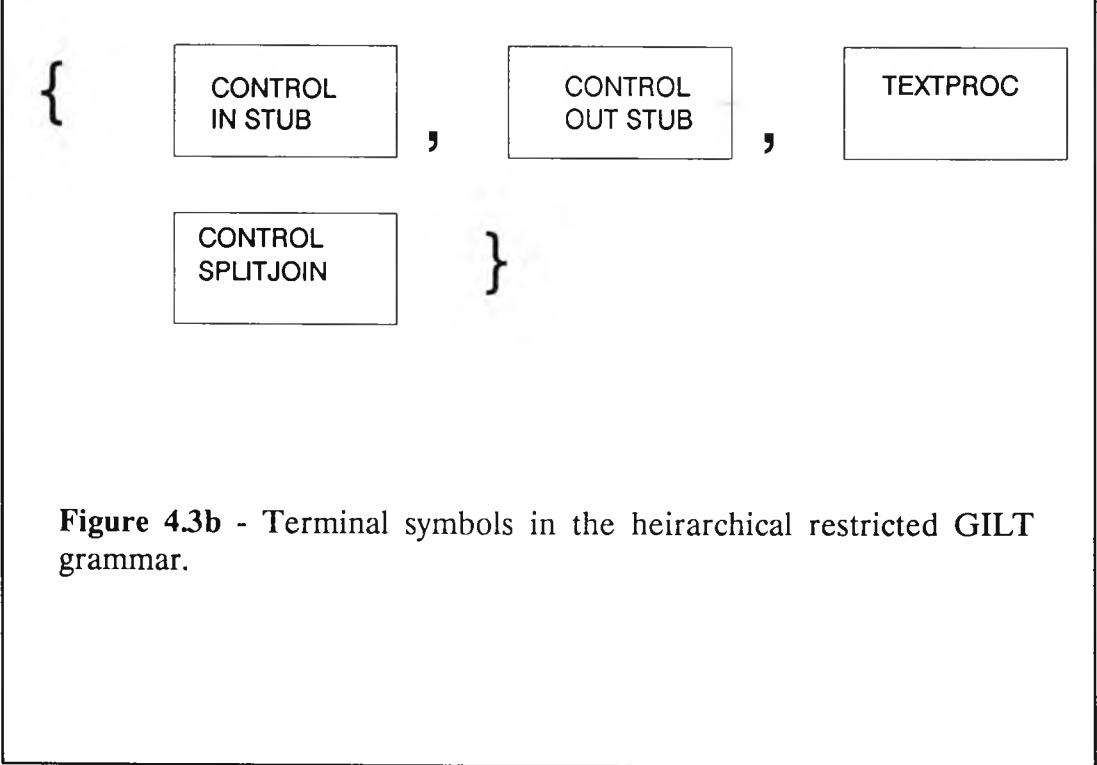


Figure 4.3b - Terminal symbols in the heirarchical restricted GILT grammar.

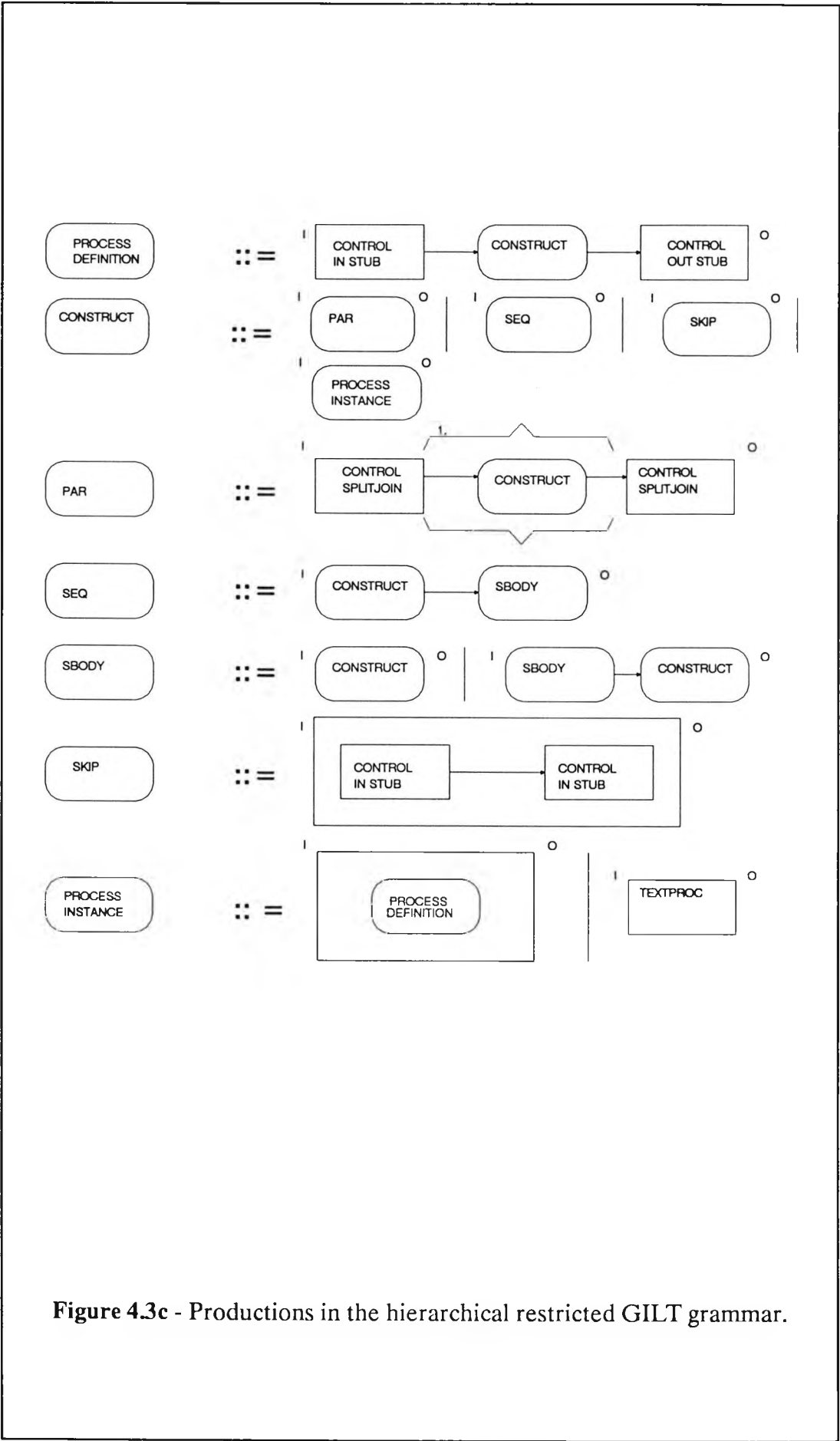


Figure 4.3c - Productions in the hierarchical restricted GILT grammar.

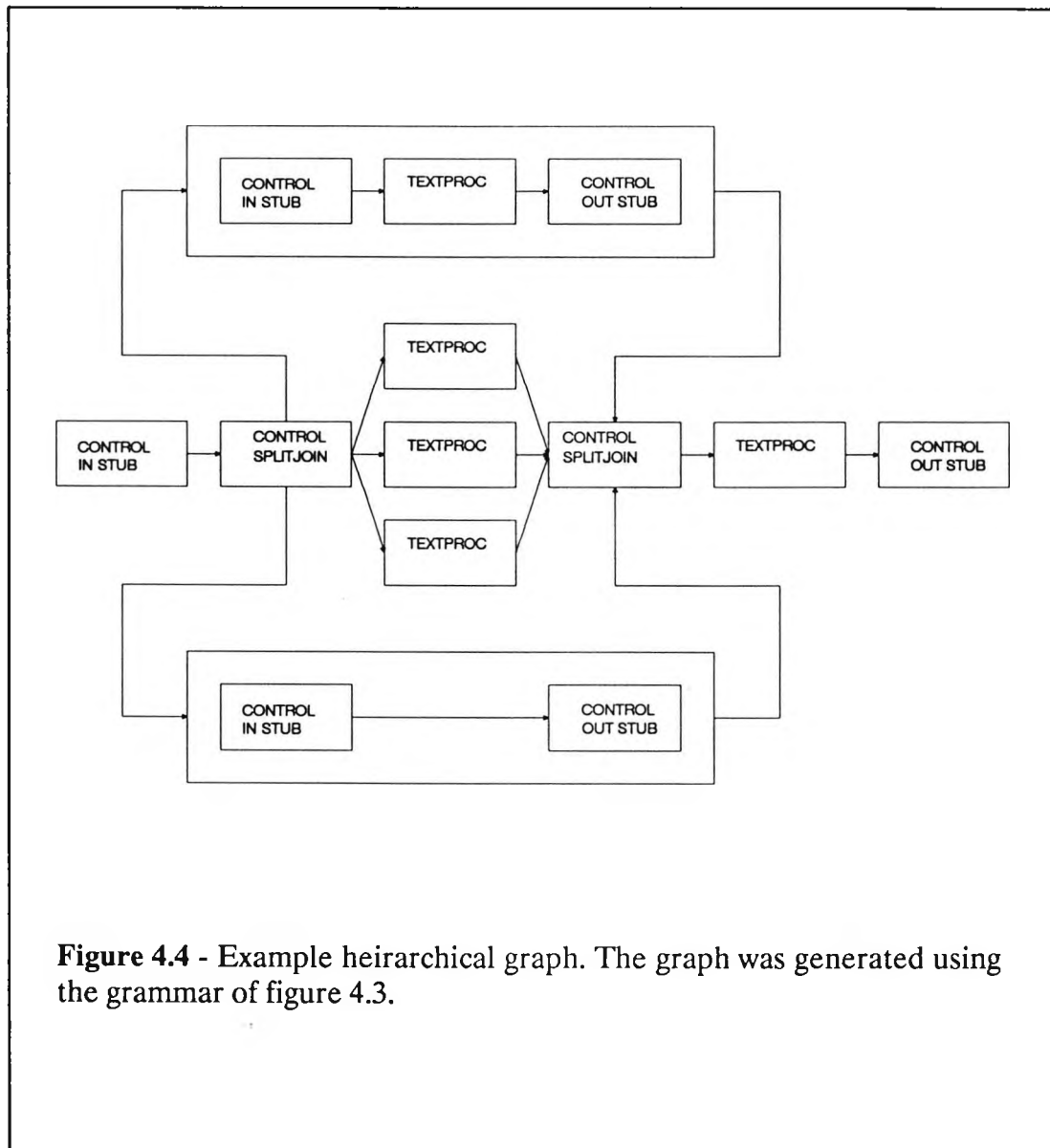


Figure 4.4 - Example hierarchical graph. The graph was generated using the grammar of figure 4.3.

than it is possible to include in a text grammar. This extra information may be quite different from that contained in conventional, textual syntactic descriptions.

As has been already shown, the control flow information in GILT may easily be represented by using a form of context free graph grammar. Unfortunately, it is not possible to include the specifications of the inter-process communications connections in the same, context free, grammar as the control flow information.

To see why this is the case, consider graphs representing processes and inter-process communication only. In such graphs, nodes represent processes, and arcs, inter-process communication. The graphs formalise the earlier bubble and arc diagrams discussed in chapter three where "bubbles" are processes, and "arcs", inter-process communication. How far can a context free grammar go towards expressing such diagrams ?

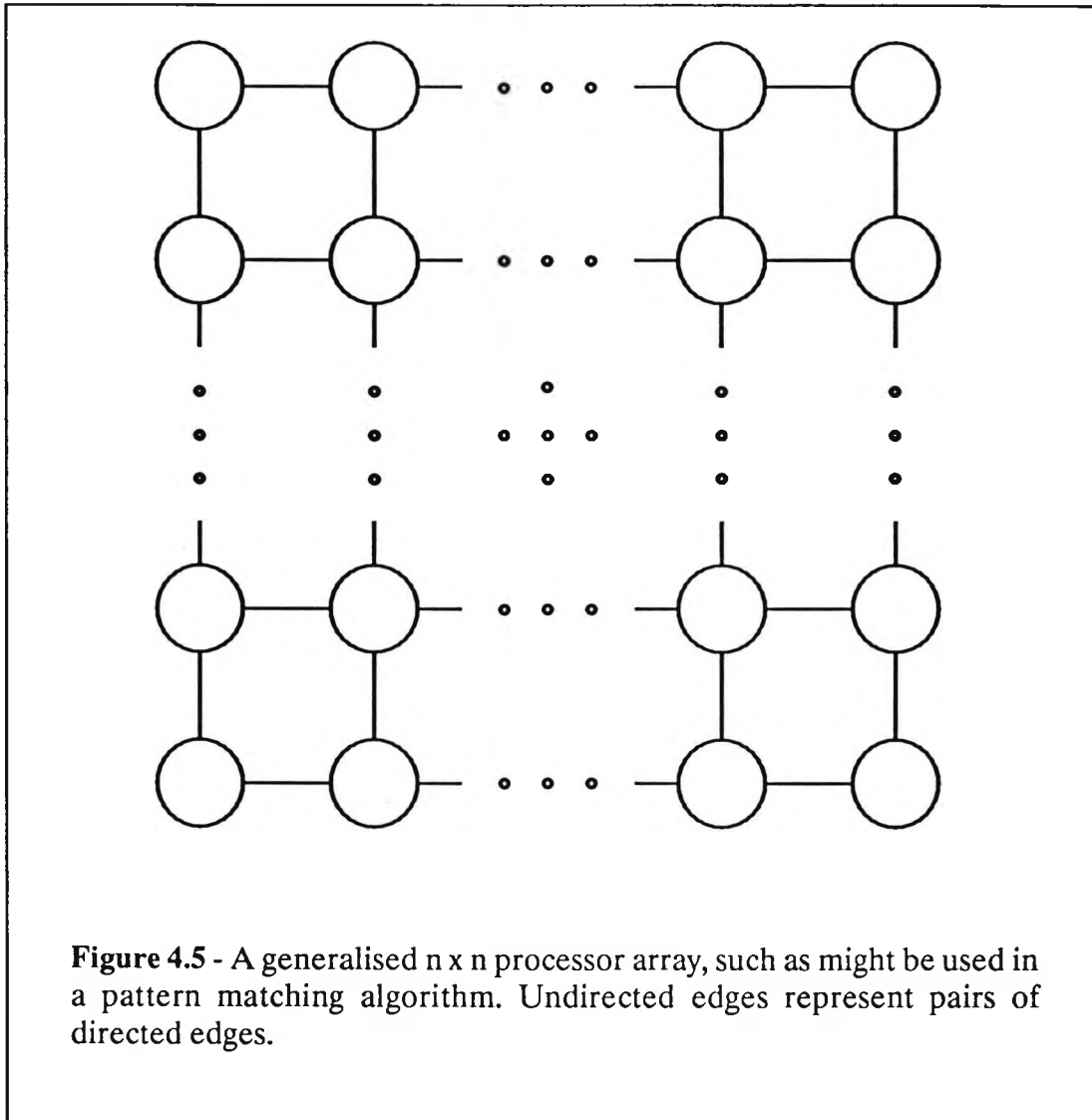
It has been shown that a context free graph grammar cannot express the complete set of planar graphs (Della-Vigna and Ghezzi, 1978) due to limitations in the embedding mechanism used by such grammars. The theorem used relies on a generalisation of the "Pumping Lemma" (Aho and Ullmann, 1972) for context free textual grammars. Productions in a context free graph grammar, like the earlier ones, must have an input ("I") and an output ("O") node which may be the same node. These nodes define the gluing points of the graph being inserted. In reverse, for a graph to be generated by a context free graph grammar, it must at least be possible to replace a non-trivial "sub-graph" with a single node. The sub-graph, which forms the right side of a production rule, must have an input ("I") and an output ("O") node (which again may be the same node). Consider the partition of a graph into a sub-graph and a "remainder graph", formed by the removal of the sub-graph from the original graph. For a graph to have been generated by a context free graph grammar it must at least be possible to find a sub-graph which has distinguished "I" and "O" nodes so that I is connected to the remainder graph only by incoming edges and O only by outgoing edges. None of the other nodes in the sub-graph may be connected to the remainder graph in any way. Clearly the sub-graph and the remainder graph must be non-trivial and have more than one node.

Amongst the planar graphs that cannot be represented are regular graphs of arbitrary size with two way links connecting between nodes. Such structures are commonly used in parallel programming, for example as regular arrays of processes used for pattern matching (see figure 4.5). They cannot be represented by a context free graph grammar because it is not possible to find a sub-graph which satisfies the conditions of the previous paragraph and has "I" and "O" nodes connected to the remainder of the graph in the correct manner. The addition of extra links and nodes to such graphs, for example those which might be required for the representation of the control flow, does not alter the nature of the problem, and hence the graphs like those required for the representation of GILT diagrams cannot be generated using a single context free graph grammar.

Several methods have been used to overcome problems like the one above. Context free productions in a grammar have been used to represent some parts of the syntax, with context dependent productions representing other parts. This approach has been used by several authors (Gottler, 1989; Nagl, 1986b; Engels 1986) and the context dependent productions in the grammar are often described informally.

Pratt (1971) suggested a scheme in which a "reduction" rule is used to amalgamate nodes with the same label thus forming an arc between them. Pratt used his approach for the representation of GOTO statements in Algol programs. This approach enables arcs to be formed between nodes generated by different rewritings, and allows the insertion of non-context free edges in graphs.

A different approach is taken here. Separate context free grammars sharing the same vocabulary are used to define different aspects of diagrams. Syntactically correct graphs are then formed by the connection of graphs defined by each context free grammar, with certain restrictions placed on the final combined graph. In



essence, different aspects of the diagrams are formalised using different grammars. This method should be easy to generalise to other visual programming languages and diagrammatic representations.

In GILT, connection points for inter-process communication connections (Channel Links) are modelled using discrete icons. The icons defining the external connections of diagrams are Channel Input Stubs and Channel Output Stubs, while their counterparts for connections to Process Icons are Channel Input Ports and Channel Output Ports. Channel connectors are used to fork and join channels so that a single channel may connect a number of different inter-process communication points, as defined by the simple graph grammar of figure 4.6.

The communication components of GILT diagrams are considered as separate from the remainder of GILT diagrams and consist of Channel Connector Icons, Channel Input Stubs, Channel Output Stubs, Channel Input Ports and Channel Output Ports interconnected by Channel Links. In the communications grammar, ports are not connected to processes but are instead considered to be separate

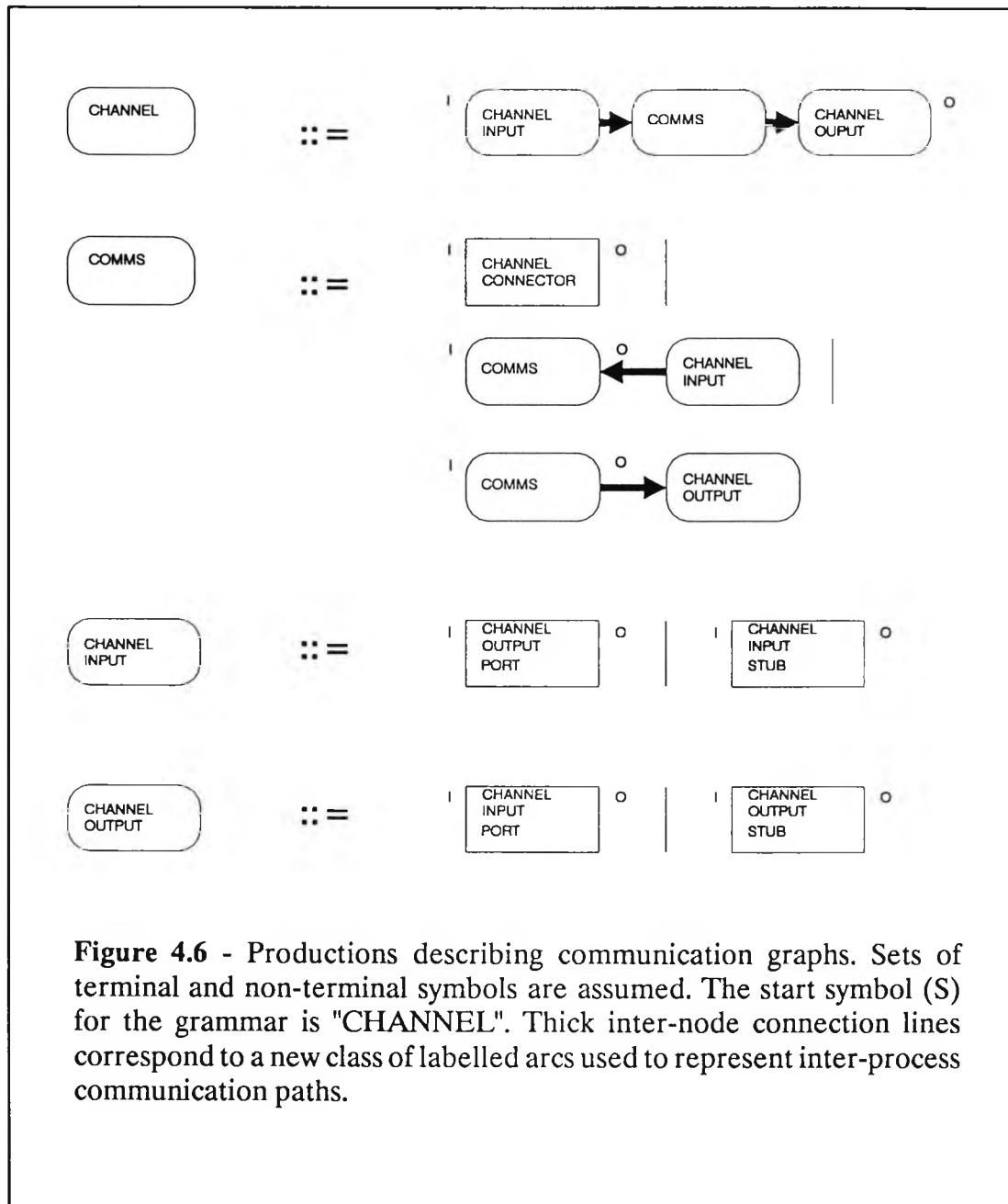


Figure 4.6 - Productions describing communication graphs. Sets of terminal and non-terminal symbols are assumed. The start symbol (S) for the grammar is "CHANNEL". Thick inter-node connection lines correspond to a new class of labelled arcs used to represent inter-process communication paths.

entities. They are subject to the following rules, which may be developed from the "communications grammar" of figure 4.6 :

- 1) No Channel Port or Channel Stub may have more than one outgoing or incoming Channel Link, though Channel Connector Icons may have multiple input and output links, and must have at least one input connection and one output link.
- 2) Channel Links can connect from Channel Output Ports or from Channel Input Stubs to Channel Connector Icons.

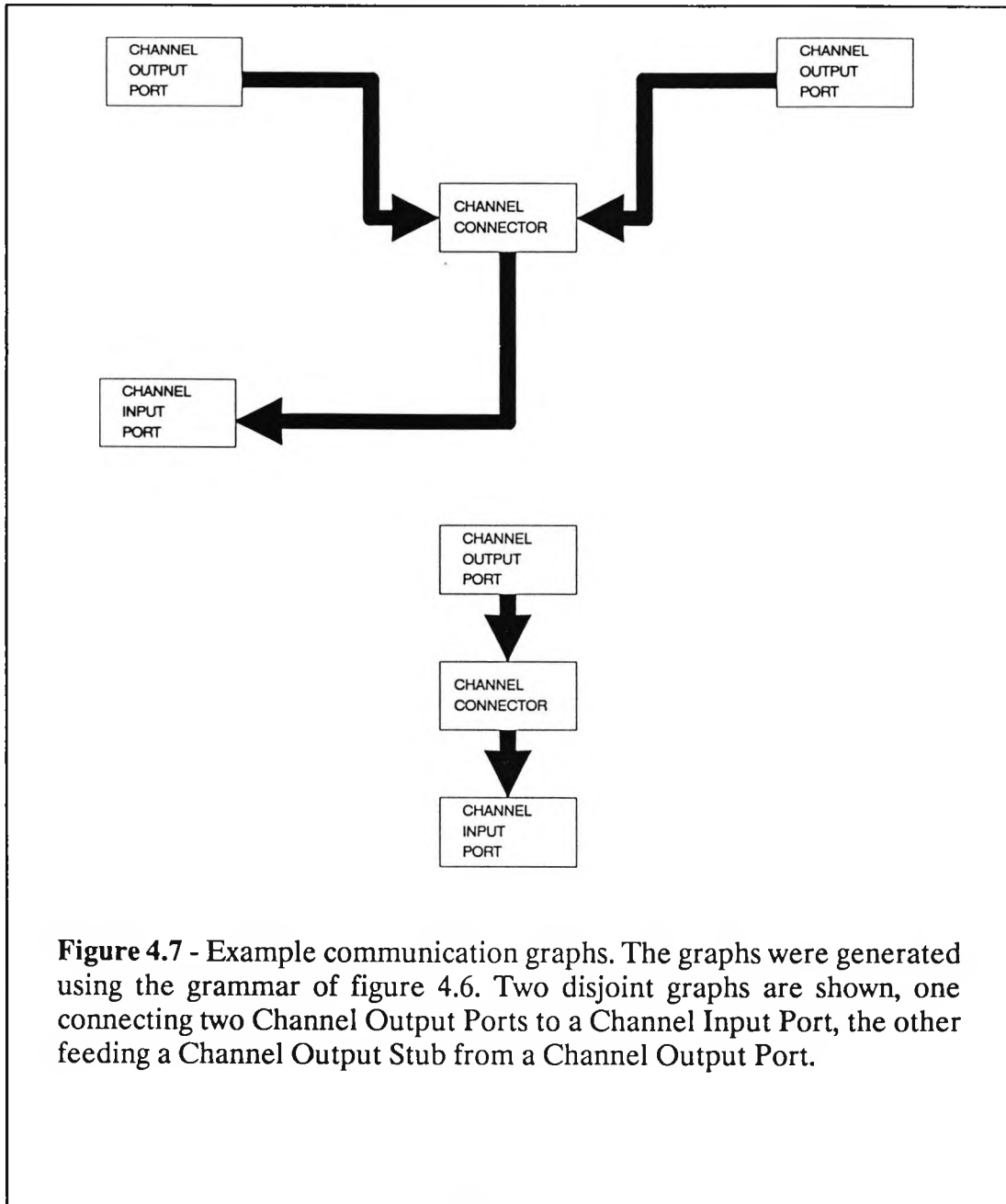


Figure 4.7 - Example communication graphs. The graphs were generated using the grammar of figure 4.6. Two disjoint graphs are shown, one connecting two Channel Output Ports to a Channel Input Port, the other feeding a Channel Output Stub from a Channel Output Port.

3) Channel Links can connect from Channel Connector Icons to Channel Input Ports or to Channel Output Stubs.

The grammar precludes the creation of communications structures whose meanings are unclear and which cannot easily be expressed in Occam (this subject is dealt with in more detail in chapter five). The representation of the communications aspects of a GILT diagram consists of zero or more disjoint "communication graphs". Each communication graph consists of a single "CHANNEL CONNECTOR" terminal connected to Channels Ports and/or Channel Stubs, as shown by the example communication graphs of figure 4.7,

which are in the set of graphs which may be generated using the communications grammar of figure 4.6.

Another grammar, called the "base grammar", may be defined in a manner similar to the earlier hierarchical simple grammar. The base grammar defined here is in fact an extension of the earlier simple hierarchical grammar of figure 4.3.

The base grammar and the communications grammar share a common vocabulary. A legal "simple GILT" graph may then be formed by the attachment of communication graphs to a base graph. The points of attachment are nodes valued with terminal values contained in productions of both the base grammar and the communication grammar.

Figure 4.8 shows productions in the base grammar defining the Process Instance non-terminal symbol. These productions replace the earlier productions for the Process Instance non-terminal of figure 4.3. The addition of the new non-terminals and terminals to figure 4.3's vocabularies is assumed. The base grammar defined for the simple GILT graphs in this chapter is simpler than the version of the grammar contained in chapter 5, which contains more productions and defines the complete language. Figure 4.9 shows an example base graph for the grammar of this chapter. The ports in the diagram have the same values as the ports in the earlier example communications diagram, figure 4.7.

A legal simple GILT graph consists of a single base graph with zero or more attached communications graphs. All Channel Port nodes in the base graph must appear in one (and only one) communications graph. A full definition of the attachment conditions is included in appendix 2.

Treating the base and communications graphs separately enables the two graphs to be separately parsed and thus reduces the complexity overall. It also allows modular changes to be made separately to the base and communications grammars.

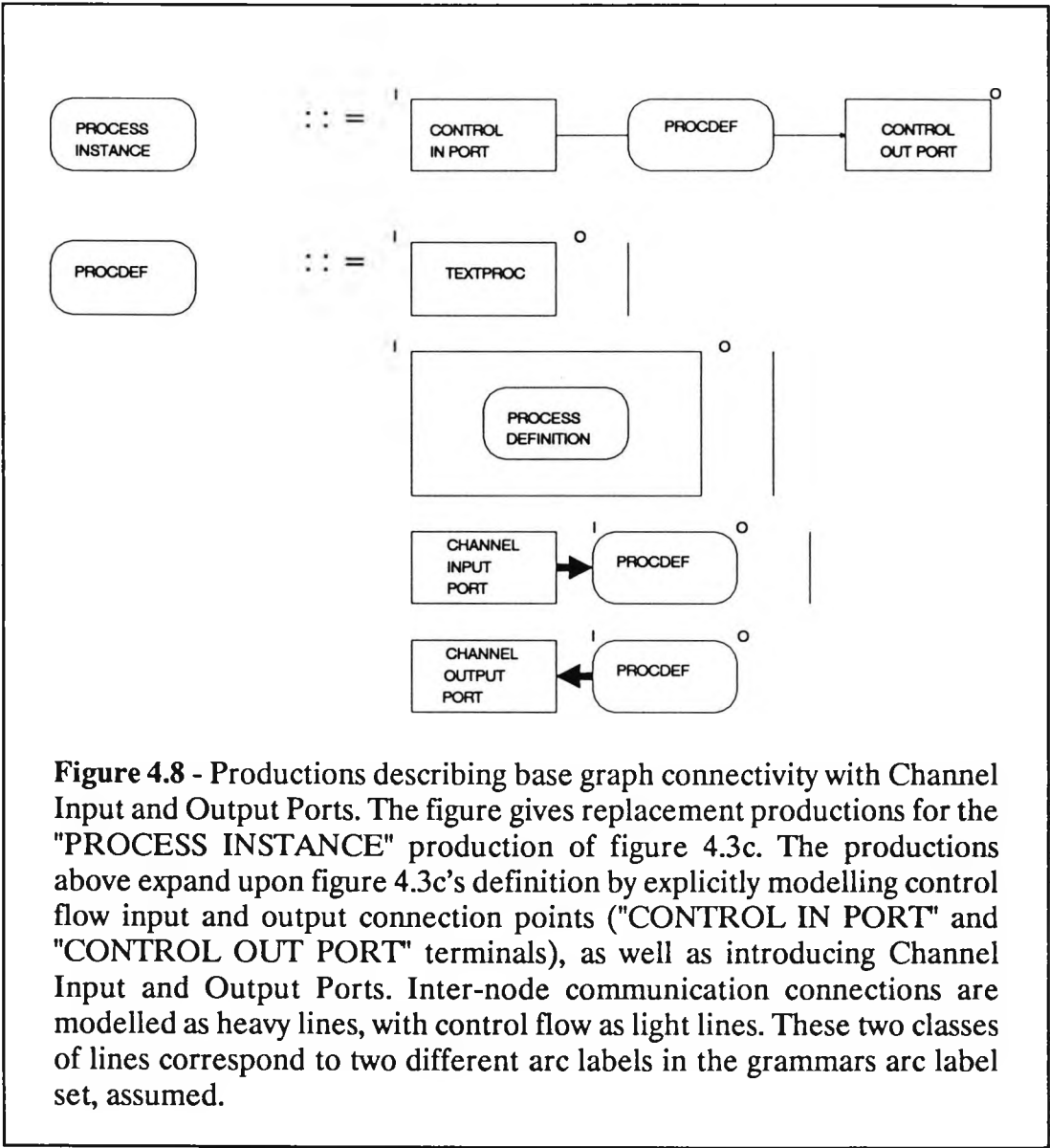
Figure 4.10 shows a legal simple GILT graph formed by the attachment of the communications graphs of figure 4.7 to the base graph of figure 4.9.

4.3.2.6 Omissions of the grammar

The two grammar system above does not implement versions of Occam's extra syntactic restrictions concerning communication between processes that are not in parallel.

An example of these restrictions, enforced by extra grammatical rules, is :

"A sequence is invalid if there is a channel which may be used only for input by one of its components, and only for output by another of its components" (Jones and Goldsmith, 1988).



which prohibit clearly erroneous code like :

```
SEQ
  ch ? x
  ch ! y
```

It is not possible to implement such restrictions in the syntax of a textual language because they are context dependent, relying on both the pattern of channel connections between processes and the pattern of control flow in the program. The idea that there might be a way of bringing such restrictions into the syntactic domain because the channel connection pattern and the flow of control within the program are modelled by graph grammars is appealing. However, the explicit nature of the channel connections and control flow in the grammars presented in

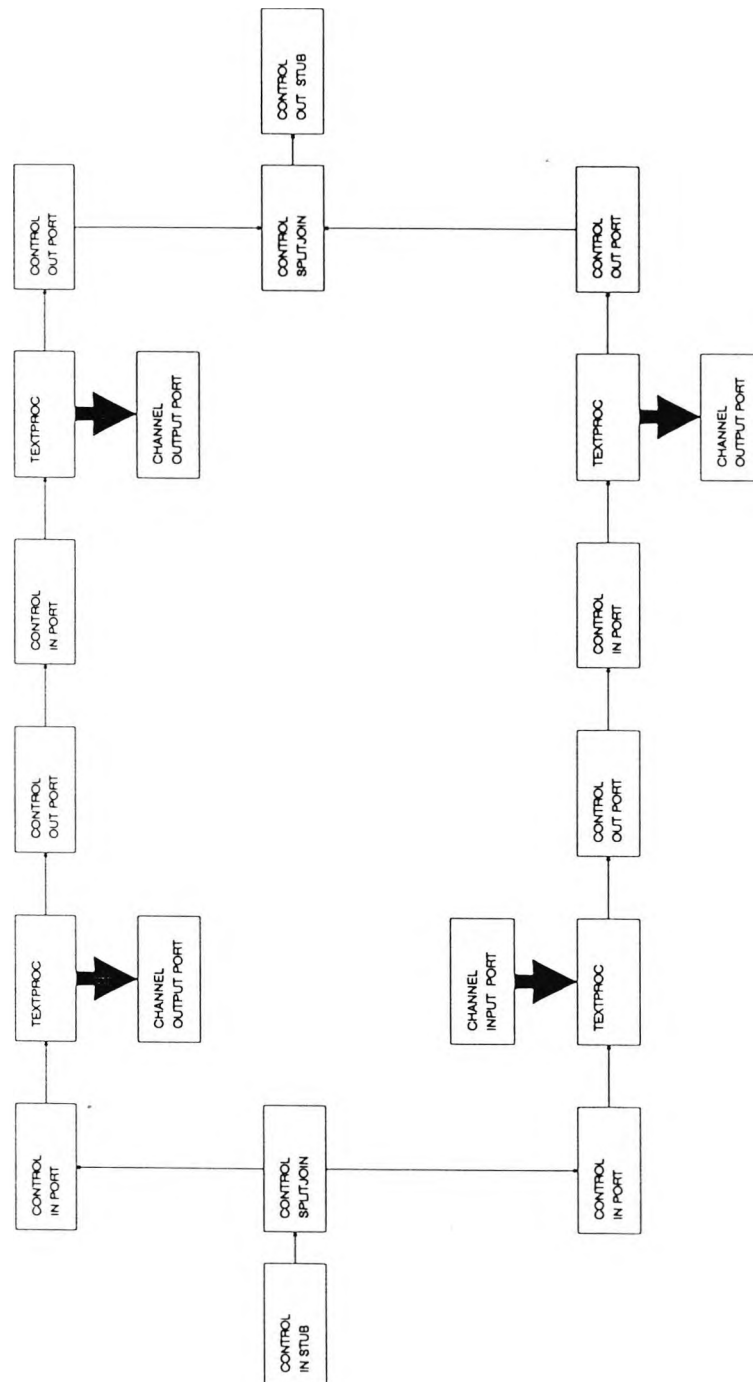


Figure 4.9 - Example base graph. A base graph generated using the simple base grammar of section 4.3.2.4.2. Connection points to the communication graph are at the nodes valued "CHANNEL INPUT PORT" and "CHANNEL OUTPUT PORT".

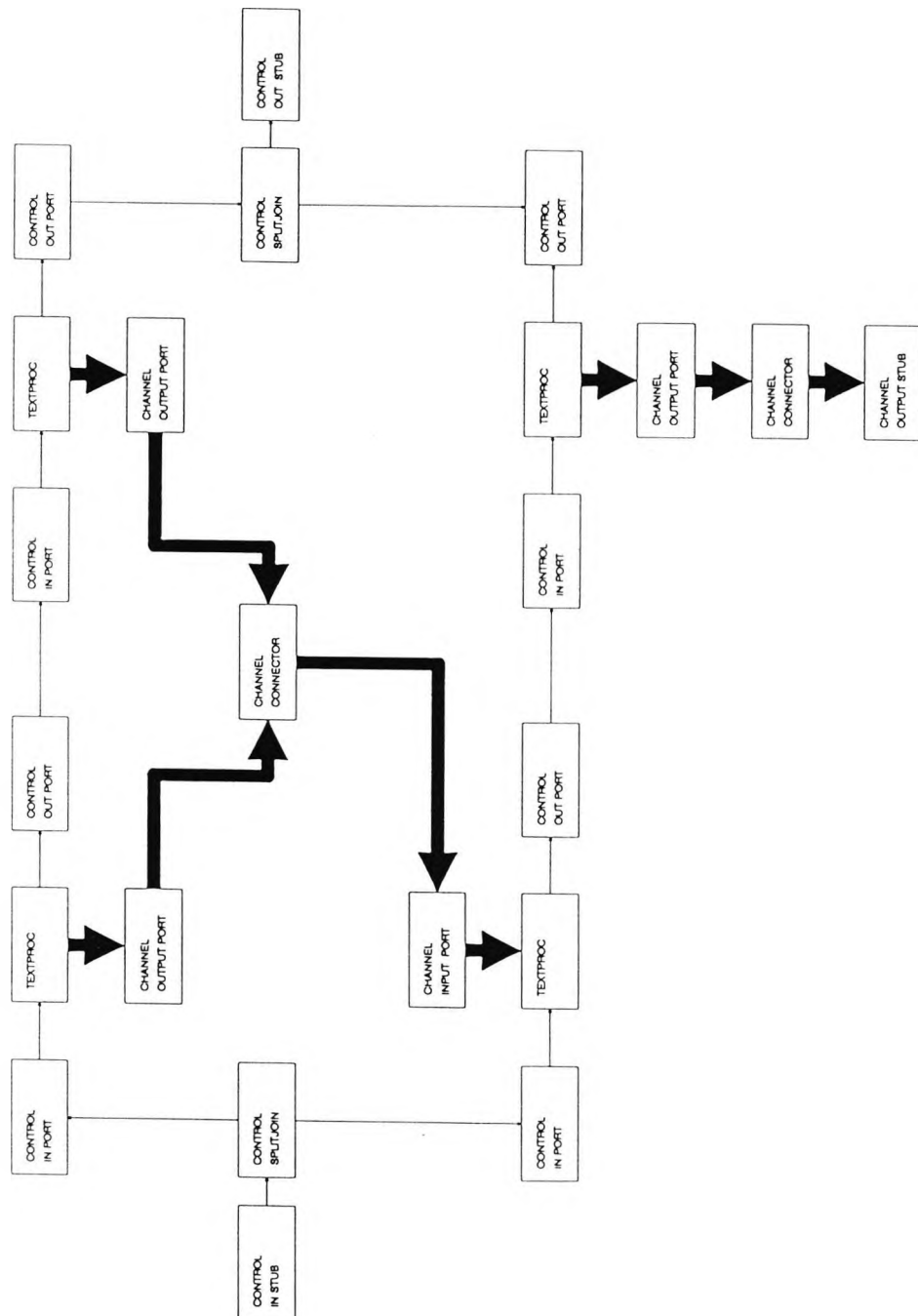


Figure 4.10 - Total graph. The graph is formed by the connection of the graphs in figures 4.7 and 4.9. Note that other graphs could be formed by the connection of the two graphs.

this chapter does not alter the context dependent nature of the problem. Therefore, restrictions like the one above may not be brought into the syntactic domain, at least for languages similar to GILT.

Extra grammatical restrictions like the one mentioned earlier are difficult to enforce formally and in any case the connection of a channel structure between processes only indicates that the processes may communicate, and not that they will communicate.

The grammars used cannot provide any form of consistency checking between the Channel Stubs in a diagram and the Channel Ports on the Process Icon which represent it. A similar situation exists for passed and declared variables. Both of these cases have a parallel in conventional sequential languages where such context dependent problems are implemented using extra syntactic rules.

4.3.2.7 GILT graph symbols and their relationship to nodes and arcs

GILT's "functional icons" correspond to terminal or non-terminal symbols in the grammar. Some functional icons (for example, "Channel Input Stubs" and "Control Output Stubs") map directly to terminal symbols in the grammar. Other functional icons (for example "Process Icons") correspond to non-terminal symbols. A complete correspondence between symbols in the grammar and GILT's functional icons is discussed in greater detail in the next chapter.

4.3.3 Describing the textual parts of the GILT language

A major facet of GILT is its ability to describe the functionality of a Process Icon using a textual or a graphical specification. Certain other components in the language (for example, variable declarations) also contain textual elements. A model for the specification of structures in a graphical specification has formed the bulk of this chapter. Grammars like those described above do not describe textual structures, with nodes containing text considered to be terminals in a grammar. Extending them to include a description of diagram components with textual elements can be regarded as a matter of converting textual BNF productions describing the textual elements to graph grammar productions in which connections between characters are explicitly modelled by arcs, as in the box language grammar. Instead of being modelled as terminal symbols, diagram components containing text are modelled as non-terminal symbols, with productions rewriting them into strings of explicitly connected characters. This work has already been carried out by (Pratt, 1971) for Algol, and presents no real challenge. Alternatively, the textual elements of diagram components may be modelled as terminal symbols. The structure of the text contained by the terminal symbols is then described by separate modified BNF style productions. This method retains the simplicity of representing the textual parts of diagrams by terminal nodes, yet allows familiar BNF productions to be used in the definition of the text's structure. As the conversion of productions from one system to another is obviously trivial the second, simpler, method is used in the following chapter.

4.4 Semantics of visual languages and the semantic definition of GILT

A formal semantic specification for a language is beneficial because it provides a precise standard for implementations of the language and can be used as a tool for language design or analysis.

Little work has been carried out on the formal expression of the semantics of visual languages. (Chang, Tortora, Yu and Guercio, 1987) proposed a formal theory of iconics in which a generalised icon has a logical part (the meaning) together with a physical part (the image). (Harel, Pnueli, Schmidt and Sherman, 1987) gave a formal semantics for statecharts, which are Hi-Graph based extensions to state transition diagrams. The latter study indicates that conventional semantic definition methods can be modified for use with visual languages.

In contrast, considerable research has been performed on the semantics of Occam. Research has taken place in three areas of semantic definition; denotational, algebraic and operational semantics. An example work from each area is quoted. Early work was involved with the production of a denotational semantics for most of the Occam1 language (Roscoe, 1984). An algebraic semantics, using a transformational approach for a similar subset was produced later (Roscoe and Hoare, 1986) while (Barret, 1988) gives an operational semantics concentrating on the communications behaviour of the Transputer implementation of Occam1. Roscoe's denotational semantic definition is currently being expanded to cover the whole of the Occam2 language (Goldsmith, 1990).

A full semantic definition of GILT is not attempted here. GILT constructs have a direct correspondence to constructs in Occam, and an informal definition of GILT's operational semantics is obtained by reference to Occam constructs. Such a definition is contained in the next chapter.

A more formal definition of GILT's semantics could be obtained by defining a formal mapping between GILT's constructs and those of Occam using, for example, a pair grammar (Pratt, 1971). Such a formal mapping would obtain a semantic definition for GILT cheaply through usage of that produced for Occam. Numerous advantages would be conferred by such a system. For example, some of the laws contained in (Roscoe and Hoare, 1986) could be considered as graph morphisms transforming between logically equivalent graphical constructs. A program transformation system for GILT based on graph transformation could therefore be produced with graph morphisms regarded as productions in a context dependent graph grammar. The effect of transformations applied to a GILT program could be viewed visually with the transformed program displayed in source, end and intermediate forms. One example application for such a system would be a sequentialisation transformation, in which parallelism is removed from a process. The transformation could be used to increase performance of a parallel

algorithm that is to be run on a single Transputer by minimising context switching. A visual program transformation system would allow the creation of a highly humanistic interface to a complex formal method, hence allowing users to manipulate programs in a natural way.

The GILT programming language

5.0 Introduction

This chapter gives a full description of the GILT language. The syntax of the language is described using the graph grammars of chapter four, while a semantic definition is obtained through reference to features of Occam. In addition, an explanation of the origins of the language's constructs and components is also included, while examples provide a demonstration of the language in use. Throughout the chapter, the first instance of a particular diagrammatic object is introduced by quotes (""), and the initial letters of each word making up an object's name are capitalised throughout. A clear distinction between the components making up GILT diagrams and other objects may therefore be made.

5.1 The GILT language

Because it supports a small number of programming constructs, the GILT visual programming language is relatively simple compared to many textual parallel languages, yet is far more comprehensive than previous visual parallel programming languages. The language's simplicity allowed a prototype implementation to be produced within a reasonable time span.

In GILT, Occam style processes are visually represented by "Process Icons" which are connected into networks with further icons and "links" of two types. The functionality of a Process Icon is defined either by a "definition diagram" ("graphical Process Icon") or by a textual specification written in Occam ("textual Process Icon"). A distinction is made between the definition of a Process Icon, a "Process Icon definition", and an instance of a Process Icon, a "Process Icon instance". This distinction is similar to the one made in textual, imperative, programming languages between the definition of a procedure and calls made to it. GILT's Process Icons are more general than textual language procedures since they represent generalised processes which may be procedural or non-procedural. Instances of "procedural Process Icons" are equivalent to calls to a procedure, while instances of "non-procedural Process Icons" are equivalent to the use of a reference to a macro definition.

In the following description of the language, the graphical facets of GILT are described prior to the textual part because GILT programs make use of the

graphical parts of the language for higher level detail, followed by the textual parts for lower level detail.

5.2 GILT diagrams

Chapter four outlined how hierarchical graph (H-graph) grammars may be used to model GILT's constructs. This chapter uses the grammars of chapter four to fully define the GILT language, whilst explaining the functionality of its various components.

GILT diagrams consist of "functional icons" which are connected into constructs by links which are attached to specific connection points. Functional icons correspond to diagrammatic symbols used in GILT's editing system and are defined in the syntax by connected networks of terminal symbols, most of which are described by productions rewriting a single non-terminal symbol, or by simple non-terminal symbols. The terminal symbols in the networks represent the components of the functional icons, "functional icon components". GILT's constructs are defined by graphs formed from terminal or non-terminal nodes representing functional icons and arcs which define the links between the components of the construct. The graphs defining the constructs are formalised in the grammar by productions rewriting non-terminal symbols into networks of arc connected nodes. As shown in chapter four, two separate sets of productions are required to describe GILT diagrams, those for a "base grammar" and those for a "communications grammar". The grammar of this chapter may therefore be thought of as having three classes of productions. Firstly, those describing functional icons, or parts of functional icons, with connection points for control flow and inter-process communication links. Secondly, productions describing control flow constructs and thirdly, productions describing inter-process communication constructs.

The components making up GILT diagrams are therefore defined and discussed in the following order, from the bottom (least complex syntactic entities) to the top (most complex syntactic entities) :

- 1) Functional icon components (defined in the syntax by terminal symbols) - section 5.2.1.
- 2) Links (defined in the syntax by labelled arcs) - section 5.2.2.
- 3) Functional Icons including Process Icon instances (defined in the syntax as connected networks of terminal symbols, most of which are described by productions rewriting a single non-terminal symbol, or by simple non-terminal symbols) - section 5.2.3.

The way in which the "diagram components" described above are used in GILT makes up the following two sections after those described above. Section 5.2.4 describes Process Icon definitions and definition diagrams, including a description of GILT's procedural and non-procedural abstraction mechanisms. Section 5.2.5

describes the constructs used in definition diagrams, giving example constructs and their uses.

5.2.1 Functional icon components

For simplicity and modularity, many of GILT's functional icons are made up of a number of discrete parts. In particular, standard connection points for the two different types of links used in the connection of functional icons into constructs are utilised in many diagram components. Each discrete part of a diagram component is modelled in GILT's grammar as a separate node. Nodes are connected by labelled arcs to form graphs which define functional icons, or parts of functional icons. Some of the simpler functional icons are described by single terminal nodes.

Functional icon components, which are modelled in the grammar by terminal nodes, are unique to specific diagram components with the exception of two classes of components ("ports" and "text areas"), which are common to many functional icons. There would be little point in discussing classes of functional icon components which are unique to particular functional icons, so a description of ports and text areas follows with the more specific components introduced in later sections as required.

5.2.1.1 Ports

Together with various other components, the five types of ports ("Control Flow Input Ports", "Control Flow Output Ports", "Not Control Flow Output Ports", "Channel Input Ports" and "Channel Output Ports") form part of GILT's larger functional icons. They were introduced into the language to provide explicit connection points for links. The use of ports separates link connection points from the other components of the functional icons, allows the structure of symbols to be modelled more accurately and reduces the complexity of the language and its editing system. It also enables an exact modular specification of component functionality to be given and ensures orthogonality between the various types of functional icons. For example, the construction of the editing system was simplified by allowing the same routines to be used as part of the behavioural specification of many functional icons.

As ports are one of the most basic components in the language, they are modelled by terminal symbols in the grammar. Figure 5.1 shows the ports as they appear in the current implementation, together with their representative terminals symbols.

5.2.1.1.1 Control Flow Ports

Control Flow Input and Control Flow Output Ports provide control flow connection sites for various type of functional icons and appear on the left and right hand sides of functional icons respectively. Control flow enters icons through

Port	Terminal symbol	Graphical representation
Control Flow Input Port	CONTROL IN PORT	
Control Flow Output Port	CONTROL OUT PORT	
Not Control Flow Output Port	NOT CONTROL OUT PORT	
Channel Input Port	CHANNEL INPUT PORT	
Channel Output Port	CHANNEL OUTPUT PORT	

Figure 5.1 - Ports in the current implementation of GILT together with their representative terminal symbols in the grammar.

Control Flow Input Ports and may exit through Control Flow Output Ports. In the present implementation Control Flow Ports are shown as 16x16 monochrome rasters depicting an appropriate directional arrow.

5.2.1.1.2 Not Control Flow Output Ports

Not Control Flow Output Ports are used in conditional control flow structures to provide a control flow output from a functional icon which is enabled when a

conditional is not satisfied (thus, "Not Control Flow Output Port"). The Not Control Flow Output Port is identified by a directional arrow like that used in the Control Output Port, but has an "N" symbol close by. Not Control Flow Output Ports are discussed in more detail in section 5.2.3.2.3.

5.2.1.1.3 Channel Ports

Channel Input and Channel Output Ports provide connection sites for "Channel Links" (described in section 5.2.2). Functionally, data items on Channel Links, which are analogous to Occam channels, may be thought of as entering icons via Channel Input Ports and exiting icons via Channel Output Ports. In the present implementation the ports consist of 16x16 monochrome rasters showing the words "IN" and "OUT" respectively. Only one Channel Link may be connected to or from each Channel Port. Channel Input and Output Ports were introduced for similar reasons to the Control Ports and provide discrete connection points for Channel Links. They are further discussed in the later section on communications constructs (section 5.2.5.2).

5.2.1.2 Text Areas

The other common functional icon components, apart from ports, are the text areas which are used for purposes such as the naming of processes and the expression of booleans in conditional control flow structures. Like ports, text areas are modelled in the grammar by terminal symbols, but may be distinguished from one another by their textual contents. The text which may be contained in a text area is defined by a few modified BNF productions like those used for Occam's grammar (appendix one).

Six types of text area are presented below; "Name Text Areas", "Expression Text Areas", "Condition Text Areas", "Shutoff Text Areas", "Input Variable Text Areas" and "Variable Declaration Text Areas". Figure 5.2 shows examples of each area, with the representative terminal symbols in the grammar. In the current implementation, each area consists of a twenty character scrollable text editing area. A smaller number of characters are displayed at any one time in the area.

5.2.1.2.1 Name Text Areas

Name Text Areas are used for Process Icon names and, in correspondence to Occam's micro syntax, each Name Text Area may contain a "name". A name is composed of a sequence of letters, digits and full stop characters, etc., the first of which must be a letter. No name may be one of Occam's reserved words.

Text area	Terminal symbol	Graphical Representation
Name Text Area	NAME TEXT AREA	<code>named.proc</code>
Expression Text Area	EXPRESSION TEXT AREA	<code>x + 101</code>
Condition Text Area	CONDITION TEXT AREA	<code>a < b</code>
Shutoff Text Area	SHUTOFF TEXT AREA	<code>TRUE</code>
Input Variable Text Area	INPUT VAR TEXT AREA	<code>invar</code>
Variable Declaration Text Area	VARIABLE DECLARATION TEXT AREA	<code>INT x</code>

Figure 5.2 - Examples of Text Areas and their representative terminal symbols in the grammar.

5.2.1.2.2 Expression Text Areas

Expression Text Areas are used as components of functional icons which pass non-channel (variable) parameters to Process Icons and may contain a variable

name or a suitable expression, as defined by the following modified BNF expression where "expression_text" is the root symbol :

```
expression_text ::= rand | mon.op rand | rand rator rand
```

```
rand ::= literal | variable | ( expression_text )
```

```
rator ::= number.op | bit.op | shift.op | relate.op
```

```
mon.op ::= - | MINUS | ~ | BITNOT
```

```
number.op ::= + | - | * | / | \
```

```
bit.op ::= /\ | BITAND | \ | BITOR | ><
```

```
shift.op ::= > | <
```

Where "literal" and "variable" are syntactic entities defined in Occam's syntax as shown in appendix 1, which also contains a definition of the notation used above.

5.2.1.2.3 Condition Text Areas

Condition Text Areas serve similar purposes to Occam's conditionals and form part of conditional control structures. The syntax of the text string which may be contained in a condition area as defined by the following modified BNF expression, where "condition_text" is the root symbol :

```
condition_text ::= expr relate.op expr
```

```
relate.op ::= equality | inequality
```

```
equality ::= = | <>
```

```
inequality ::= < | <= | > | >=
```

"expr" is equivalent to "expression_text" is defined in the previous section.

The productions define a subset of Occam's boolean conditional expression. A subset is used to exclude Occam features not supported in GILT diagrams, such as function calls.

5.2.1.2.4 Shutoff Text Areas

Shutoff Text Areas are similar to Conditional Text Areas, but are used in the guards of alternative constructs. Structurally, any text contained in a Shutoff Text Area may be the same as that contained in a conditional, but the text area of a shutoff may also be empty, so that a shutoff area does not have to contain any text:

```
shutoff_text ::= empty | condition_text  
empty ::=
```

The root symbol is "shutoff_text".

5.2.1.2.5 Input Variable Text Areas

Like Shutoff Text Areas, Input Variable Text Areas are also used in the guards of alternative construts. Structurally, the area may be empty or contain a reference to an integer variable :

```
input_variable_text ::= empty | variable  
variable ::= name | variable[int.expr]
```

where int.expr is syntactically equivalent to the "expression_text" symbol above, but is restricted to having an integer value as it is used for array subscripting. The root symbol is "input_variable_text".

5.2.1.2.6 Variable Declaration Text Areas

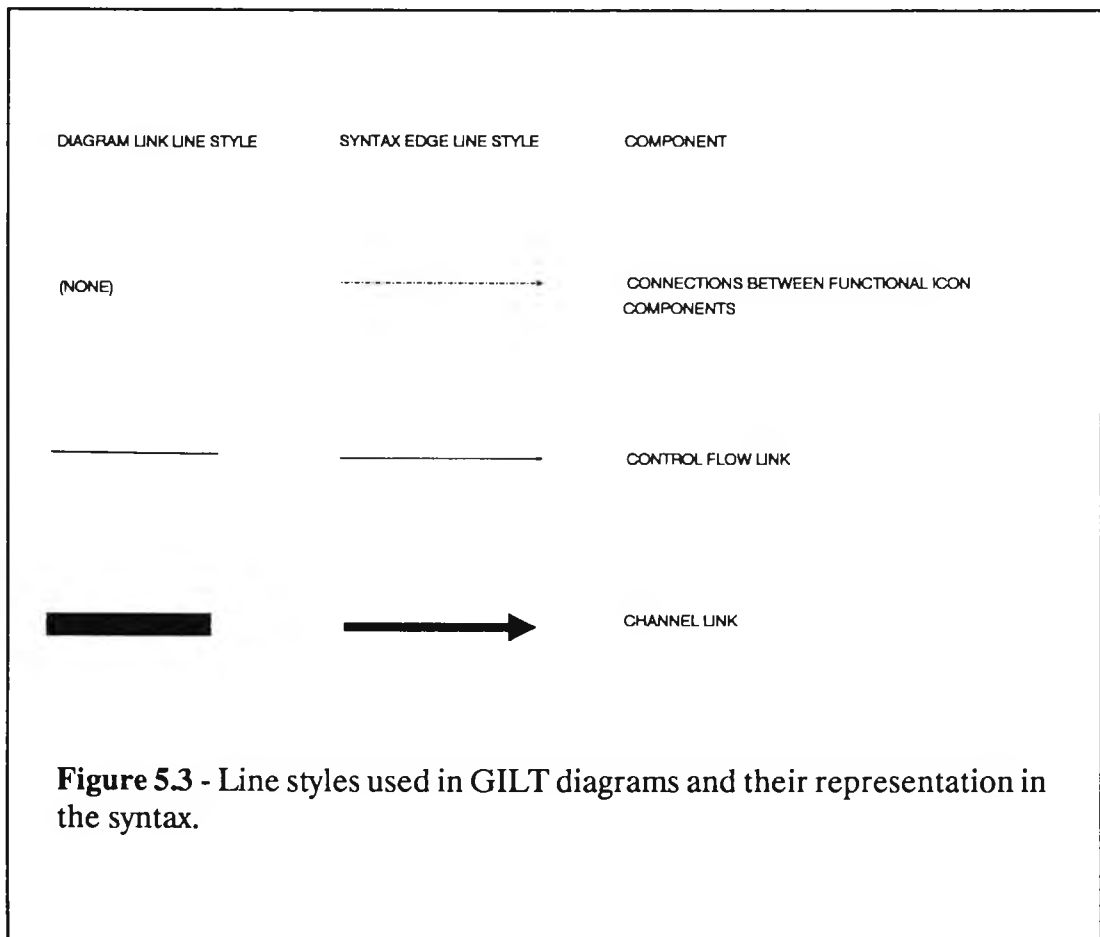
Variable Declaration Text Areas form parts of two different functional icons; those used for variable declarations and those used in the declaration of non-channel parameters for Process Icons. The symbol "variable_declaration_text" is the root symbol :

```
variable_declaration_text ::= specifier name  
specifier ::= base.type | [expr.option] specifier  
base.type ::= BOOL | BYTE | int.type | float.type  
int.type ::= INT16 | INT32 | INT64 | INT  
float.type ::= REAL32 | REAL64  
expr.option ::= int.expr | empty
```

The symbol "int.expr" (defined earlier) is used for array sizing, while "name" is a name as defined in section 5.2.1.2.1 and is used to refer to a defined variable or parameter. The syntax used is similar to Occam's, but due to limitations on screen size it excludes Occam style multiple variable definitions (separated by commas) and some of Occam's expression evaluation features not supported by GILT at the graphical level of abstraction.

5.2.2 Links

Connections between components in GILT's diagrams are made with flexible links which were chosen because their common use in electronic circuit design systems meant that they were well known to many potential system users. Two types of link exist in GILT diagrams - "Channel Links" and "Control Flow Links". Channel Links are used for the specification of possible inter-process communication pathways, while Control Flow Links specify the execution order of processes and wire together functional icons into constructs analogous to Occam constructs. Different line styles are used to differentiate between Channel Links and Control Flow Links, which are represented in the syntax by labelled arcs. Labelled arcs are also used to represent the logical connections between the functional icon components making up each functional icon. These arcs thus have no direct diagrammatic counterpart in GILT's diagrams, except that all of the components of a particular functional icon are physically close to each other and are easily recognised as being part of the same diagrammatic object. Instead of writing a textual label on each arc in GILT's production, which would be confusing, different line styles are used to differentiate between differently labelled arcs. The correspondence between labelled arcs in the syntax and the links in the diagrams is shown in figure 5.3.



5.2.3 Functional icons

Functional icons have various purposes, such as aiding the delimitation of constructs and the representation of processes. They may be divided into two main classes; those forming parts of constructs and those that do not form parts of constructs. All functional icons are uniquely distinguishable from one another and are defined in GILT's grammar by networks of connected terminal nodes, or by simple terminal nodes. The set of functional icons is shown in figure 5.4 together with their corresponding grammatical representations. Most functional icons are represented in the grammar by non-terminal symbols with associated productions rewriting the non-terminals into the networks of connected terminal symbols mentioned above. The structure of any particular functional icon may be found by looking up the appropriate non-terminal(s) from figure 5.4 in the productions of figure 5.5. Figure 5.5 gives a set of productions for the base grammar of the full GILT language.

Functional icons may be divided into two classes - those that form part of constructs, and those that do not (specifically functional icons for comments and for local variable declarations). Functional icons which do not form part of constructs are discussed first, followed by a section on functional icons which do.

5.2.3.1 Functional Icons which do not form part of constructs

This class of functional icons correspond to those to which Control Flow Links or Channel Links may not be connected. The most basic functional icon in this class is the Comment. There are also Variable Declaration Icons.

5.2.3.1.1 Comments

Comments are simple text strings which may be positioned anywhere on the screen. They have no connections for Control Flow or Channel Links, and are ignored by the compiler. In the grammar, Comments are modelled by the terminal "COMMENT". Any number of Comments may be placed in a definition diagram. Comments were introduced to allow additional annotation of GILT diagrams in line with GILT's philosophy of a mixed textual/graphical paradigm. It has been noted that few other visual programming systems have allowed such textual comments (Myers, 1988).

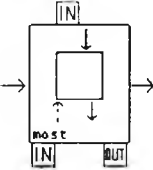
FUNCTIONAL ICON NAME	GRAMMAR SYMBOL	GRAPHICAL REPRESENTATION
Comment	COMMENT	Requests for data enter here
Variable Declaration	VARIABLE DECLARATION	Var: [S]INT buf
Process Icon Instance	PROCESS INSTANCE	
Control Input Stub	CONTROL IN STUB	CI
Control Output Stub	CONTROL OUT STUB	CO
Channel Input Stub	CHANNEL INPUT STUB	IN
Channel Output Stub	CHANNEL OUTPUT STUB	OUT

Figure 5.4a - The equivalence between functional icons and their grammatical representations.

FUNCTIONAL ICON NAME	GRAMMAR SYMBOL	GRAPHICAL REPRESENTATION
Condition Icon	CONDITION NOT CONTROL OUT PORT	
Guard Icon	GUARD	
Declared Parameter Icon	DECLARED PARAMETER	
Passed Parameter Icon	PASSED PARAMETER	
Channel Connector Icon	CHANNEL CONNECTOR	
Control Split Join Icon	CONTROL SPLIT JOIN	

Figure 5.4b - The equivalence between functional icons and their grammatical representations.

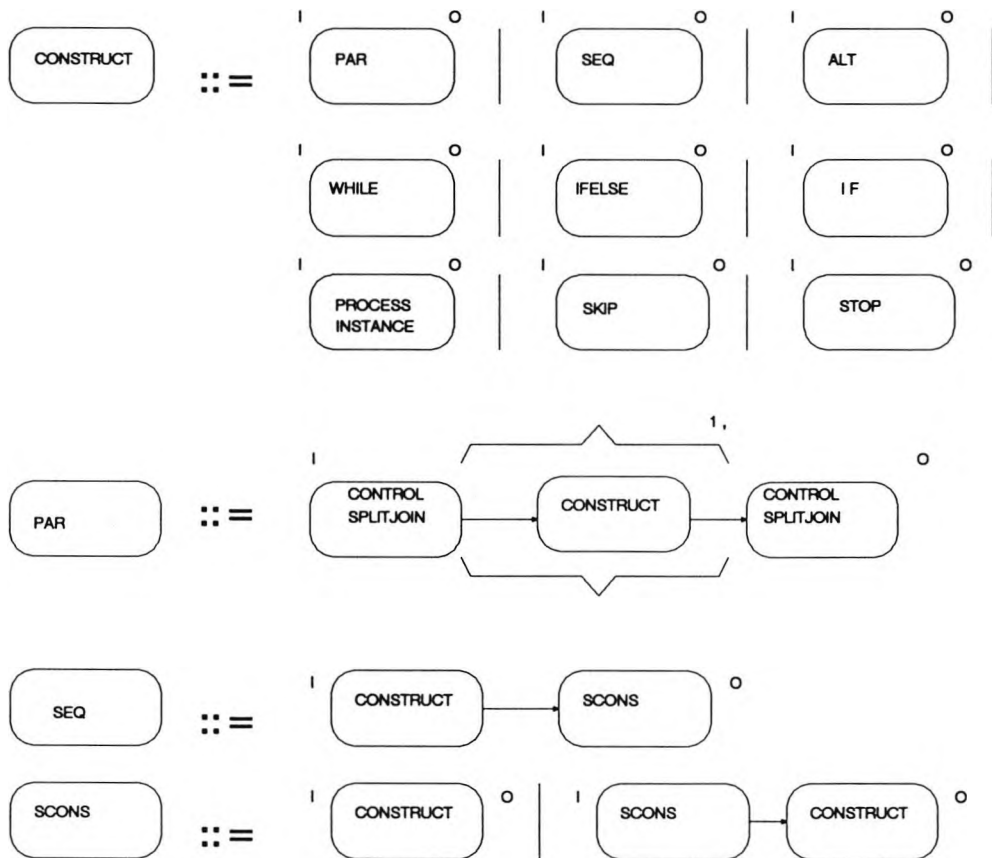


Figure 5.5a - The productions in the base grammar describing the GILT language. Terminal symbols are enclosed in rectangles with square corners, while non-terminal symbols are shown enclosed by rectangles with rounded corners. Also shown is the set of labelled arcs used between components. Sets of terminal and non-terminal symbols are assumed, and are not shown.

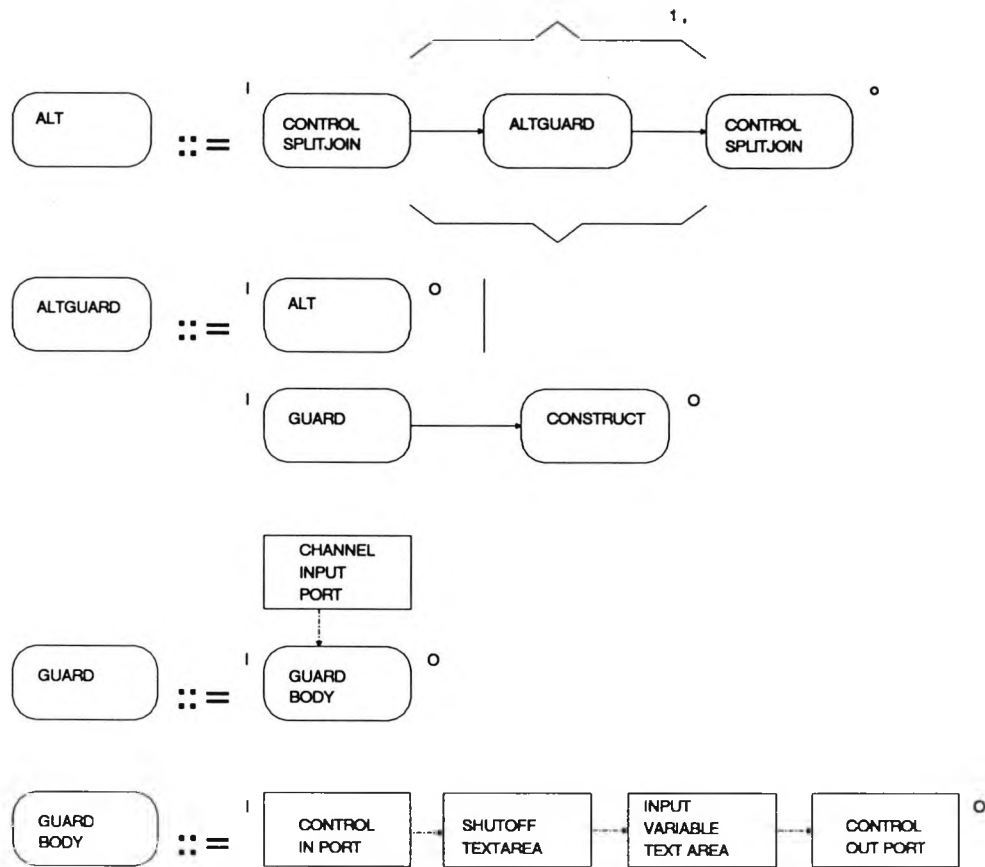


Figure 5.5b - The productions in the base grammar describing the GILT language, continued.

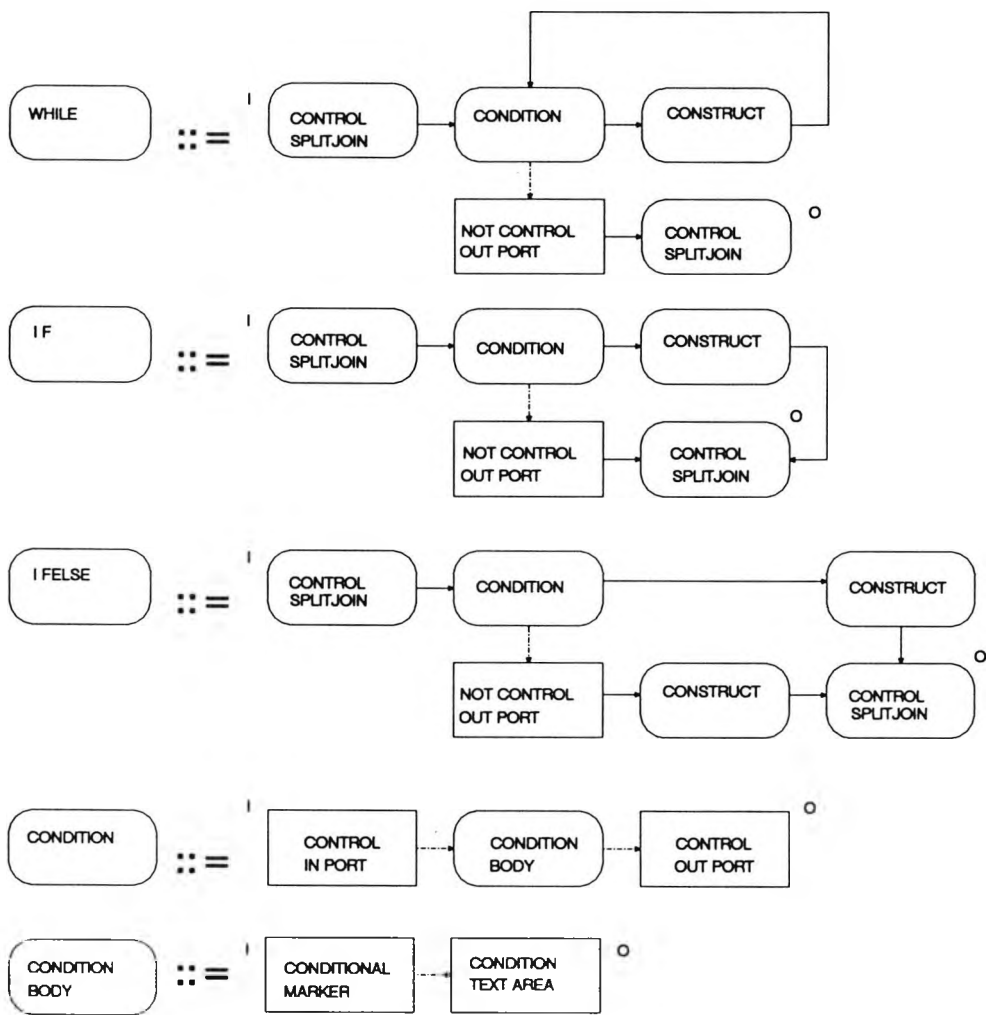


Figure 5.5c - The productions in the base grammar describing the GILT language, continued.

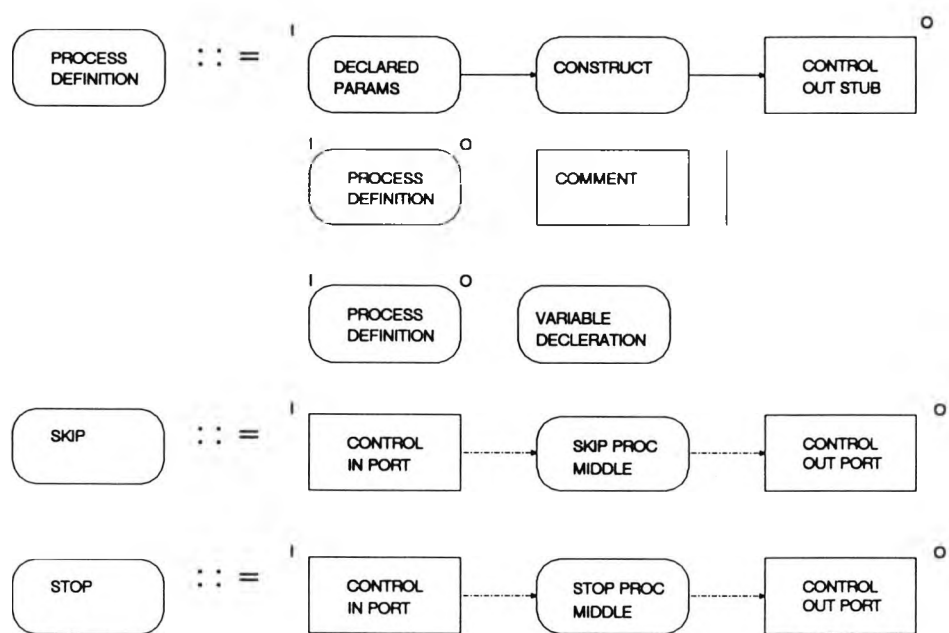


Figure 5.5d - The productions in the base grammar describing the GILT language, continued.

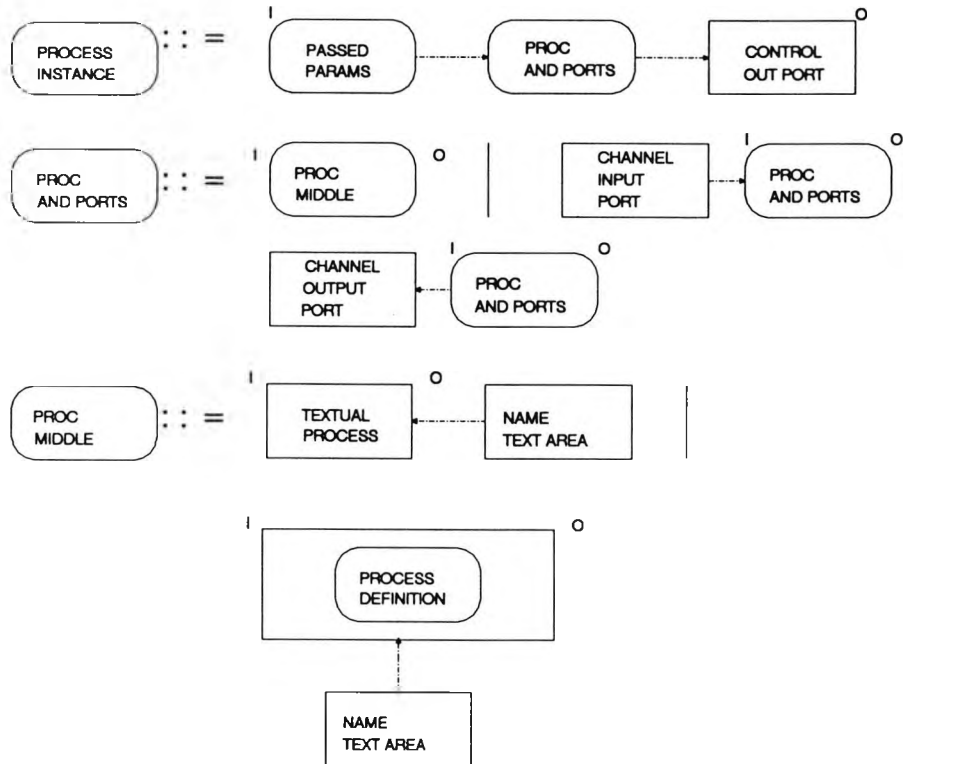


Figure 5.5e - The productions in the base grammar describing the GILT language, continued.

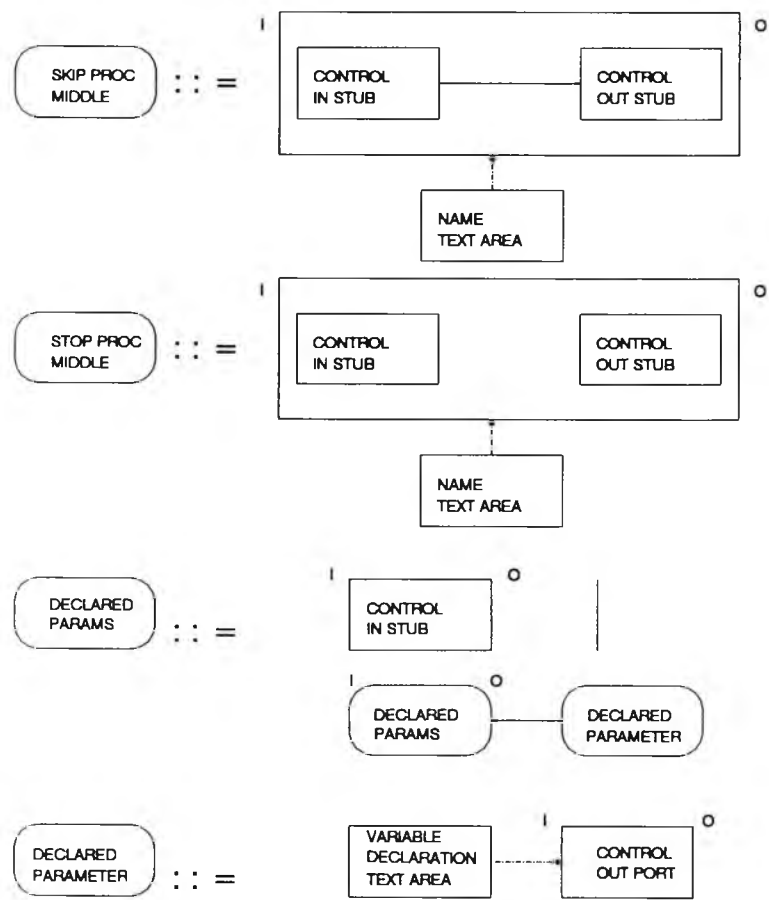


Figure 5.5f - The productions in the base grammar describing the GILT language, continued.

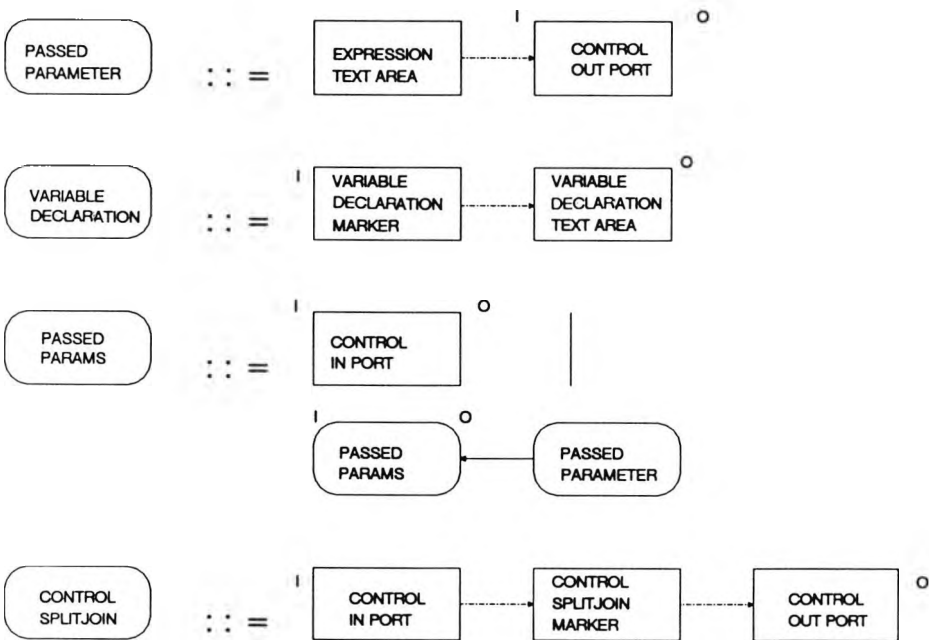


Figure 5.5g - The productions in the base grammar describing the GILT language, continued.

5.2.3.1.2 Variable Declaration Icons

Variable Declarations Icons are visually similar to Comments but have an additional "Var" string as a distinguishing label. They allow local variables to be declared at a graphical level of abstraction. Declared variables are considered to be outside the flow of control in a particular diagram and the variables defined by them hold scope over and "downwards" from the diagram in which they are defined. Such variables may be used as passed parameters for Process Icons. Alternatively, they may be used by non-procedural Process Icons at equal or lower levels of abstraction than that at which their Variable Declaration Icon occurs. Variables may have as their type any of the Occam types and array declarations, including multi-dimensional arrays, are allowed. As with Comments, as many Variable Declaration Icons as are required may be included in a definition diagram. Variable Declaration Icons are modelled in GILT's grammar by a non-terminal valued "VARIABLE DECLARATION".

5.2.3.2 Functional Icons which form parts of constructs

The Functional Icons in this class may have Control Flow Links and/or Channel Links connected to them and may thus be connected into constructs. As the purpose of many functional icons is interwoven with the semantics of the constructs of which they form part, descriptions of the structural aspects of functional icons are given here with semantic definitions following in the later section on constructs.

5.2.3.2.1 Process Icon instances

A distinction between Process Icon definitions and Process Icon instances has already been made (section 5.1). Process Icon instances are the most fundamental of the functional icons used in GILT diagrams, consisting of a small (64 x 64) monochrome raster with an associated eleven character text label which is surrounded by ports for the connection of Channel Links and Control Flow Links. The raster and the contents of the text label are derived from the Process Icon's definition, which is described in section 5.2.4. The textual label (which provides a name for the Process Icon instance) is modelled in the grammar by a Name Text Area. No two Process Icon definitions within the same program may have the same name, though multiple Process Icon instances may exist in the same way that multiple references to a procedure are allowed in textual languages. On the left side of a Process Icon instance is a Control Flow Input Port and on the right side is a Control Flow Output Port. At the top and bottom of the icon are as many Channel Input Ports and Channel Output Ports as are defined by the Process Icon's definition. Process Icons are unique amongst the components of GILT graphs in that their functionality and, to a degree, their appearance is defined by the user. All other symbols have system defined functionality and appearance.

Earlier versions of GILT (Roberts and Samwell, 1989) used Process Icons without textual names. Textual names were subsequently added (Roberts and Samwell, 1990) because there are many circumstances where the functionality of a process cannot be well described using an iconic representation without a textual label. Icons with textual labels retain the "instant recognition" capabilities of "pure" icons, but in addition have the capability to display further relevant information.

Process Icons may be procedural (equivalent to Occam procedure definitions and references) or non-procedural (similar to macros in textual languages). The type of a Process Icon is determined by the use of a visual "toggle" whose value is associated with the Process Icon's definition. The semantics of procedural and non-procedural Process Icons is discussed in a later section of this chapter (5.2.4), with the visual toggle discussed in chapter six. Instances of both types of Process Icon may have passed parameters attached to their Control Input Port. The structure of a Process Icon instance and its passed parameters is modelled in the grammar by the non-terminal "PROCESS INSTANCE".

5.2.3.2.2 Stubs

Stubs provide the definition of a diagram's external connectivity. They provide similar facilities to ports, but do not form part of other icons. A stub in the definition diagram of a Process Icon appears as a port on an instance of a Process Icon. Stubs are essential to GILT's visual abstraction and folding mechanism, having a similar function to pads in VLSI design systems. GILT's stubs are modelled by terminal symbols in its grammar.

5.2.3.2.2.1 Control Input and Control Output Stubs

Control Input and Control Output Stubs provide connections for Control Flow Links to higher levels of abstraction. All GILT diagrams have one Control Input Stub and one Control Output Stub and no more than one Control Input Stub and one Control Output Stub are allowed per diagram. The Control Input Stub defines the position at which the flow of control enters the diagram while the Control Output Stub defines the position at which control exits the diagram. The stubs appear as Control Input Ports and Control Output Ports on instances of the Process Icon defined by the diagram containing the stubs. They were introduced to provide explicit incoming and outgoing points for the connection of Control Flow Links and to give support for "skip" and "stop" constructs based on Occam's SKIP and STOP processes. Control Input and Output Stubs are modelled in the grammar by the terminal symbols "CONTROL IN STUB" and "CONTROL OUT STUB" respectively.

5.2.3.2.2.2 Channel Input and Output Stubs

Channel Input and Channel Output Stubs look like bigger versions of Channel Input and Output Ports, and can be seen in figure 5.4. They provide connections

for Channel Links to higher levels of abstraction and are similar to the definition of passed channels in an Occam procedure header. Data items enter a GILT diagram from Channel Input Stubs and exit via Channel Output Stubs. In the present implementation they are shown as 64x64 monochrome rasters showing the words "IN" and "OUT" respectively. Channel Input and Output Stubs provide obvious connection points for incoming and outgoing Channel Links, and are modelled in the grammar with terminal symbols "CHANNEL INPUT STUB" and "CHANNEL OUTPUT STUB".

5.2.3.2.3 Condition Icons

Condition Icons form part of GILT's "if", "if..else" and "while" constructs. They consist of a Conditional Text Area with an associated visual label provided by a small raster with a diamond shaped image. A Control Flow Input Port, Control Flow Output Port and Not Control Flow Output Port are provided for the connection of Control Flow Links. Condition Icons provide control flow switching facilities for the construction of conditional control flow structures. They are modelled in the grammar by the non-terminal "CONDITION" with an attached "NOT CONTROL FLOW OUTPUT PORT" terminal node. Condition Icons are the only functional icons which are not described in GILT's grammar by a single non-terminal symbol and associated productions. The Condition Icon and the Not Control Output Port cannot be combined into a single non-terminal node with an associated production due to limitations in the embedding mechanism used by the graph grammar, which does not allow more than one distinguished output gluing point to be specified in each production. Representing a Condition Icon as a single non-terminal node would require two such points, introducing additional complexity into the grammar's embedding mechanism. Functionally, control enters a Condition Icon at its Control Input Port and exits at either the Control Output Port or the Not Control Output Port dependent upon whether the textual condition held by the Condition Text Area is true or false.

Other possibilities for the representation of conditional flow of control included the use of conditioned arcs similar to those used in state transition diagrams. This course was rejected as it would have introduced extra complexity into the language, the grammar and the language's editing system.

5.2.3.2.4 Guard Icons

Guard Icons provide support for alternative selection processes by modelling Occam guard statements. Guards consist of a connected structure formed from a Control Input Port, a shutoff area, a Channel Input Port, an input variable area and a Control Output Port. The semantics of a Guard Icon may not be discussed without reference to the alternative construct designed to contain it, and is left until a later section (5.2.5.1.6). Guard Icons are modelled in the grammar by the terminal "GUARD".

5.2.3.2.5 Declared Parameter Icons

Declared Parameter Icons are used to define non-channel parameters for Process Icons. They consist of a Variable Definition Text Area (for defining the type and name of a non- channel (variable) parameter) which is attached to a Channel Output Port and allows the Declared Parameter Icon to be linked to a Control Input Stub. Information on the use of Declared Parameter Icons is included in section 5.2.4. Declared Parameter Icons are modelled in the grammar by the non-terminal "DECLARED PARAMETER".

5.2.3.2.6 Passed Parameter Icons

Passed Parameter Icons are used to instantiate the parameters defined by Declared Parameter Icons. They consist of an Expression Text Area attached to a Channel Output Port which allows the linking of the passed variable to the Control Input Port of a Process Icon. Further information on Passed Parameter Icons may be found in section 5.2.4. Passed Parameter Icons are modelled in the grammar by the non-terminal "PASSED PARAMETER".

5.2.3.2.7 Channel Connector Icons

Channel Connector Icons allow the convergence and divergence of Channels from a single point. They are used as syntactic sugar to enforce certain rules regarding the connection of Channel Links in the language. Further information may be found in section 5.2.5.2. Channel Connector Icons are modelled in the grammar by the terminal "CHANNEL CONNECTOR".

5.2.3.2.8 Control Split Join Icons

Control Split Join Icons divide and combine the control flow in GILT diagrams. They consist of a Control Flow Input Port, a Control Flow Output Port and a small distinguishing icon sandwiched between the two. Control Split Join Icons enable forking and combining of control flow in the language and are used to delimit the start and end of conditional constructs. They were introduced to allow GILT to be fully modelled using a context free graph grammar, with the use of a single symbol for both forking and combining control flow, avoiding the use of two symbols. Control Split Join Icons are modelled in the grammar by the non-terminal "CONTROL SPLIT JOIN".

Early versions of GILT (Roberts and Samwell, 1989) did not contain Control Split Join Icons. Instead, parallel control flow constructs were produced by forking control flow out of Control Flow Output Ports or Input Stubs. Control flow was combined at Control Flow Input Ports or Output Stubs. This arrangement worked

well, but it was not possible to represent all of the control flow structures of Occam efficiently. For example, a representation of two parallel constructs wired in series, equivalent to :

```
SEQ
  PAR
    process1
    process2
  PAR
    process3
    process4
```

required the introduction of an extra Process Icon in addition to those required for the representation of the four processes, in order to combine and fork the flow of control between the two parallel constructs. Figure 5.6 shows an example diagram using this style of representation. Far better is the diagram of figure 5.7 which shows control flow explicitly forking and joining via Control Split Join Icons. Using the Control Split Join Icon, users can place processes and wire up control flow wires to produce legal structures, just like using a CAD package. A good analogy is building a network of resistors wired in series and in parallel.

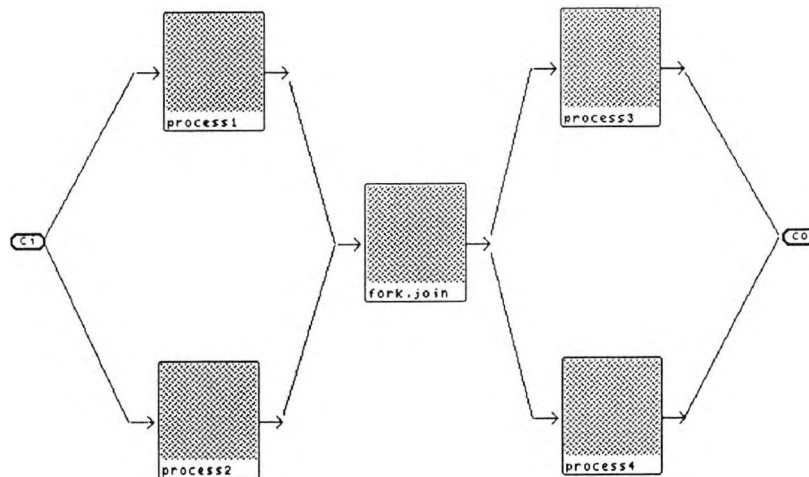
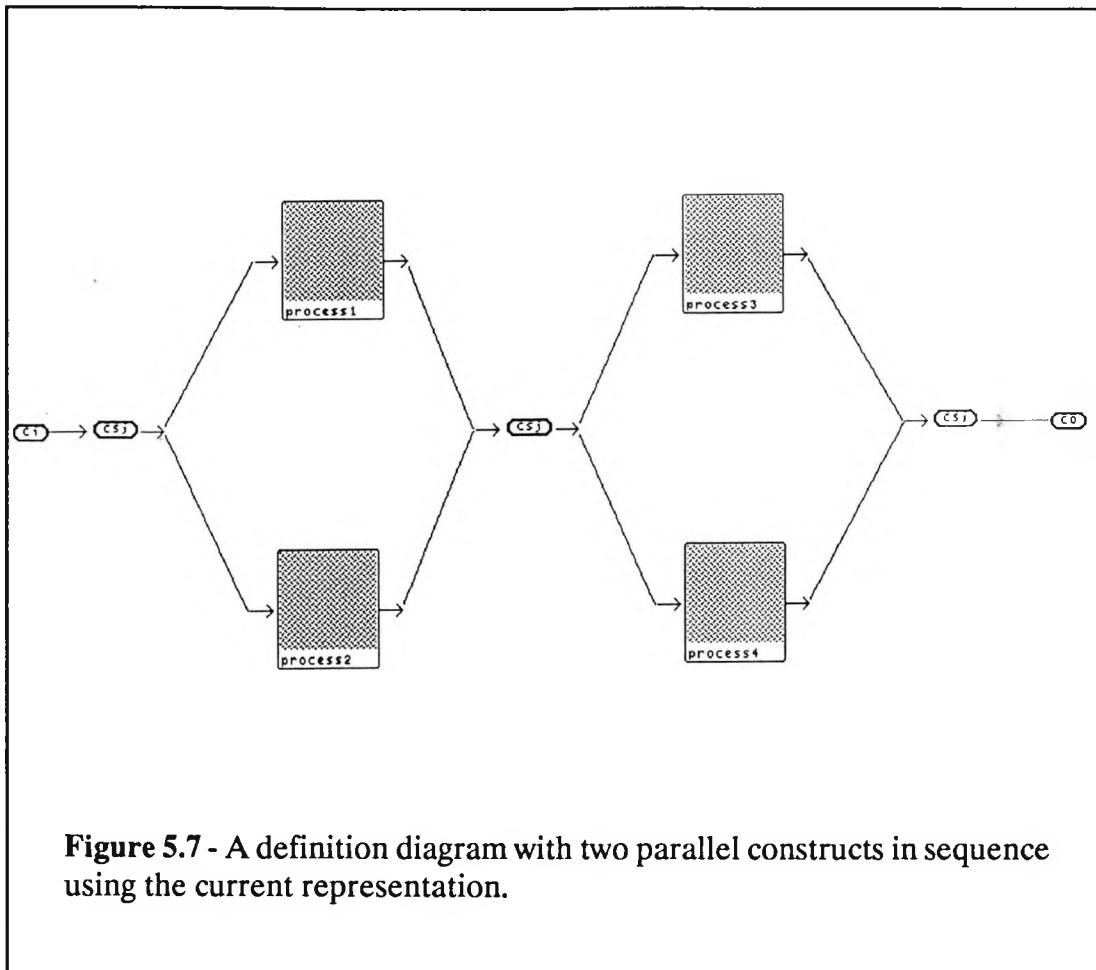


Figure 5.6 - A definition diagram with two parallel constructs in sequence using the original representation.



5.2.4 Visual abstraction and the definition of Process Icons

A distinction has already been made between the definition of a Process Icon (a Process Icon definition) and instances of a Process Icon (Process Icon instances). A Process Icon definition for a graphical Process Icon consists of two parts; firstly, a diagram defining the functionality of a Process Icon referred to as a definition diagram and secondly, a "Process Icon representation", which consists of a small black and white raster with an associated process name. A Process Icon instance is visually depicted using the raster and name from the Process Icon representation, and is surrounded by Ports for the connection of Channel Links and Control Flow Links, as discussed earlier.

A definition diagram is defined in the grammar by the non-terminal "PROCESS DEFINITION". Rewritings of the non-terminal "PROC MIDDLE", which is used to describe the central area of a Process Icon instance, allow nested definition diagrams to any required depth to be developed. The distinction between Process Icon definitions and Process Icon instances is a purely notational one which allows multiple instances of the same Process Icon. It may be thought of as providing "pointers" from Process Icon instances to Process Icon definitions, and is not enforced in the grammar, which treats all Process Icon instances separately.

Two icons must be present in every definition diagram - a Control Input Stub and a Control Output Stub. These stubs define the entry and exit points for control flow in the diagram, providing connection points for Control Flow Links to constructs defined in GILT's base grammar. Some of the most basic constructs in GILT are constructed using these two components, which are also used in the declaration of non-channel parameters for a Process Icon definition.

As discussed in the introduction to this chapter, Process Icon definitions may be of two types, procedural or non-procedural. The distinction between procedural and non-procedural Process Icons is made by the programmer through use of the program editing system, which is discussed in chapter six. Procedural Process Icons are compiled as such, with references generated as necessary. Non-procedural Process Icons are compiled as inline code.

GILT diagrams rely on the principle of visually passing parameters to Process Icon instances and visually declaring parameters for the Process Icon definitions which define the functionality of the instances. The semantics of the parameter passing process is determined by the procedural or non-procedural nature of the Process Icon definition for which parameters are defined. Parameters for a Process Icon definition are declared with Channel Input and Output Stubs (for channel parameters) and Declared Parameter Icons (for non-channel parameters). Parameters are passed to a Process Icon instance by the connection of Channel Links to the Process Icon's Channel Input Ports (incoming channels), from its Channel Output Ports (outgoing channels) and by the connection of Control Flow Links from Passed Parameter Icons to the icon's Control Flow Input Port (non-channel parameters).

Declared non-channel parameters are described in the grammar by the non-terminal node "DECLARED PARAMETERS". Similarly, passed non-channel parameters are described in the grammar by the non-terminal "PASSED PARAMETERS". Passed and declared channel parameters are part of GILT's communications constructs, described in section 5.2.5.2.

For procedural Process Icons, a Process Icon definition is equivalent to the declaration of an Occam procedure. Instances of procedural Process Icons connected in GILT diagrams are equivalent to references to an Occam procedure, while Channel Input Stubs and Channel Output Stubs in the definition diagram of a procedural Process Icon are the equivalent of passed channel declarations in the Occam procedure definition. Similarly, Declared Parameter Icons connected with Control Flow Links to the definition diagram's Control Flow Input Stub are the equivalent of non-channel parameter declarations in the Occam procedure definition.

For non-procedural Process Icons, a Process Icon definition is similar to the definition of a macro in a conventional textual language. Instances of non-procedural Process Icons connected in GILT diagrams are equivalent to references to a macro definition. In compilation, all passed parameters are unified with the Process Icon's declared parameters. Non-procedural Process Icons provide all the facilities of folds in the Inmos Transputer Development System and

allow hierarchical structuring of programs without the use of procedural abstraction. In addition however, they have the extra functionality given by macros.

There is a one to one correspondence between Channel Stubs in a definition diagram (which define channel parameters) and the Channel Ports on a Process Icon instance (which are used to pass channel parameters). Each Channel Stub in the definition diagram has an equivalent Channel Port on the Process Icon instance. The positions of the Channel Ports on a Process Icon are determined by the positions of the Channel Stubs in the Process Icon's definition diagram. A simple algorithm which quantises the screen into eight areas is used to determine the position of a port on a Process Icon from its corresponding stub in the Process Icon's definition diagram. Stubs above the centre of the diagram correspond to ports above the central area of the Process Icon instance while those below the centre correspond to ports below the central icon area. Similarly, four columns, each of which is a quarter the size of the diagram, are used to assign the left - right position of the ports on the Process Icon instance. For purely practical reasons (limited screen resolution and size) only eight stubs (and thus ports) are allowed per Process Icon. This restriction could easily be overcome with the use of a bigger raster for the representation of the Process Icons, or smaller rasters for the representation of ports, and presents no real problems. When an instance of a defined Process Icon is connected into a diagram, Channel Links connected to Channel Ports on the instance correspond to passed channels in a reference an Occam procedure. Figure 5.8 shows an instance of a Process Icon, without connected links, illustrating the correspondence between stubs and ports.

A similar scheme to that used for channel connections is used for passing and defining non-channel parameters. Declared Parameter Icons connected to the Control Flow Input Stub of a definition diagram provide the non-channel parameters of the Process Icon defined by the definition diagram. They correspond to the declaration of variable parameters for an Occam procedure. When an instance of a defined Process Icon is connected into a diagram, Passed Parameter Icons connected to the Control Input Port of the Process Icon define the calling parameters for the instance. They correspond to passed variables in a reference to an Occam procedure. Calling non-channel parameters are matched to declared non-channel parameters in a manner similar to that used in conventional textual languages. In such languages, calling variables are matched to declared variables by positioning in the relevant lists. GILT's system does not use a list position technique (there is no "list" of parameters!). Instead, it relies on the relative y-axis position of the non-channel parameters. Thus, the "highest" calling non-channel parameter is matched to the "topmost" declared non-channel parameter, and so on. Figure 5.9 illustrates the non-channel parameter passing mechanism.

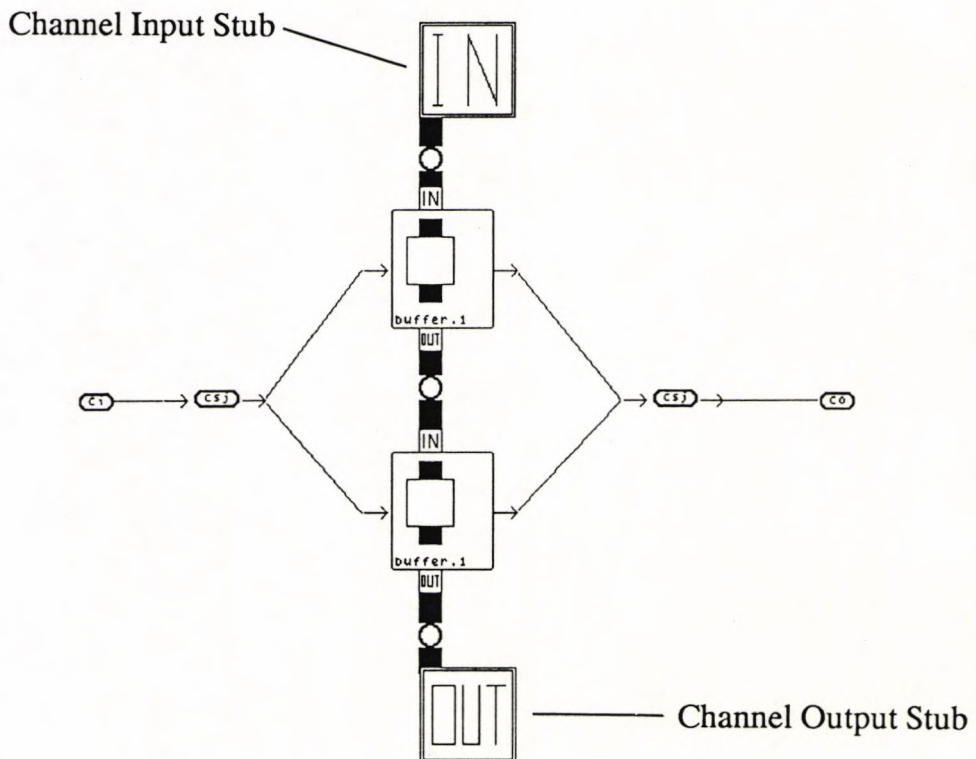
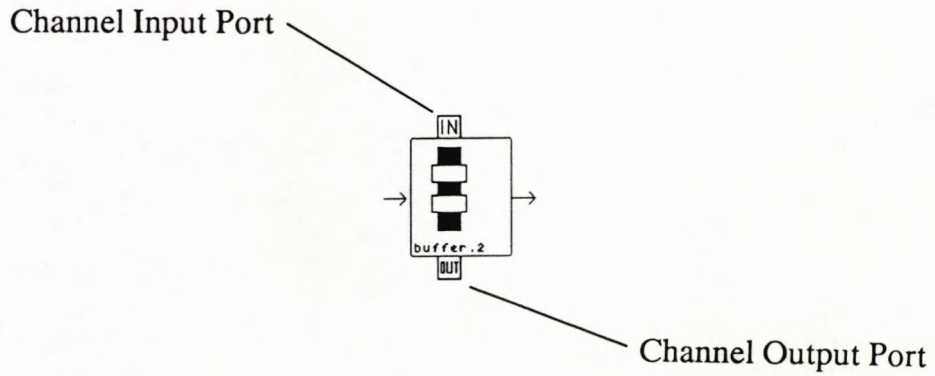


Figure 5.8 - A Process Icon instance and its definition diagram, showing the correspondence between stubs and ports.

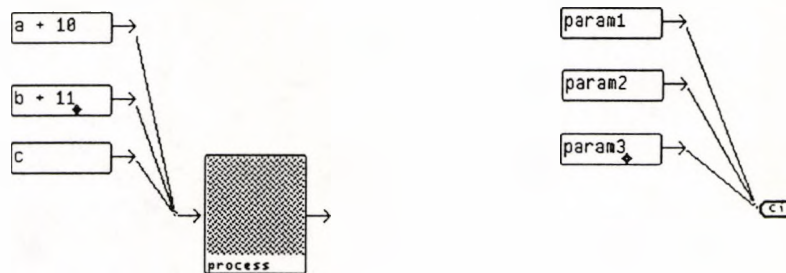


Figure 5.9 An illustration of GILT's non-channel parameter passing mechanism. The left hand side shows a number of Passed Parameter Icons connected to the Control Flow Input Port of a Process Icon, while the right side shows a portion of the Process Icon's definition diagram showing the defined non-channel parameters. The topmost parameter on the left hand side instantiates the topmost on the right hand side, with the next lowest left instantiating the next lowest right, etc.

5.2.5 Constructs

Constructs in GILT are formed by the connection of functional icons by links. Control flow constructs are formed by the connection of functional icons by Control Flow Links, while inter-process communication constructs are formed by the connection of Channel Links between functional icons. Control flow constructs are discussed initially, followed by communications constructs.

5.2.5.1 Control flow constructs

Control flow constructs in GILT are described using productions in GILT's base grammar (figure 5.4). Each control flow construct is represented by an alternate rewriting of the non-terminal node "CONSTRUCT". Only functional icons having Control Flow Ports are connected together via Control Flow Links to form control

flow constructs. In general, the functionality of a graphical Process Icon definition is specified by a construct which is connected between a Control Flow Input Stub and a Control Flow Output Stub, which must be present in every definition diagram. There are two exceptions, both concerned with the modelling of the Occam SKIP and STOP processes. In these exceptions, the Control Flow Input and Output Stubs are used to form very simple constructs.

GILT has only a small number (9) of control flow constructs ("stop", "skip", "unconstructed process", "sequence", "parallel", "alternative", "if", "if..else", and "while"), but provides all the elements essential to a parallel, imperative, programming language. Each construct has an equivalent in Occam.

GILT's constructs are described in the following sections according to the ordering given above. Each description includes an informal semantic and syntactic definition of the construct it discusses. References to the non-terminal symbols used in the grammar to represent the constructs are also included.

Many descriptions make reference to the "start" and "end" of constructs. The start and end points of a construct are defined by the input ("I") and output ("O") gluing points for the production describing the construct.

5.2.5.1.1 The stop and skip constructs

The simplest possible definition diagram consists of a Control Input Stub and a Control Output Stub only. Without any specification of the flow of control within the diagram, the control "stops" at the Control Input Stub, and never proceeds. The process represented by this diagram is equivalent to the Occam STOP process. Figure 5.10 shows GILT's stop construct, defined by the non-terminal "STOP" in the grammar.

The next simplest definition diagram is formed by the connection of the two stubs by a Control Flow Link. Such a definition diagram is semantically equivalent to the Occam SKIP process. Figure 5.11 shows the GILT skip construct, defined by the non-terminal "SKIP" in the grammar.

Both constructs provide a natural visual expression for Occam's STOP and SKIP constructs with control flow seeming to behave like an electrical current faced with an open or a short circuit.

5.2.5.1.2 Unconstructed process

One of the most basic elements in GILT is a simple (single) Process Icon instance, known as an unconstructed process. Unconstructed processes are used as components of constructs in the same way that Occam's primitive processes are used in the construction of processes, and are represented by the non-terminal "PROCESS INSTANCE" in the grammar.



Figure 5.10 - The stop construct in GILT. On the left is a Control Input Stub, with a Control Output Stub on the right. No connection between the two ports is shown. Hence control enters the structure, and never proceeds.



Figure 5.11 - GILT's skip construct. Like the stop construct it has a Control Input Stub and a Control Output Stub, but it has an additional Control Flow Link between them indicating that control flow passes straight through the definition diagram without action.

5.2.5.1.3 Sequential construct (sequence)

A sequence or sequential construct is formed by the connection of a number of constructs end to start via Control Flow Links and may be thought of as a "chain" of constructs. The sequence behaves like the Occam sequence and runs the first

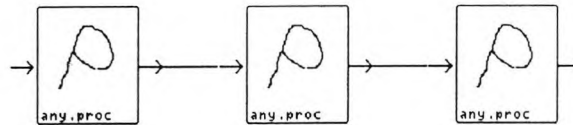


Figure 5.12 - An example sequential construct. Three instances of the Process Icon "any.proc" are shown, wired one after another.

construct until it terminates, then running the rest in sequence. As in Occam, GILT's sequence is associative in that a sequence may be formed by joining together two sequences. Figure 5.12 shows an example sequential construct. Sequential constructs are represented by the non-terminal "SEQ" in the grammar.

5.2.5.1.4 Parallel construct

A parallel construct is formed by the connection of a number of constructs in parallel. Control Flow Links from the Control Flow Output Port of a single Control Fork Join Icon are connected to the start of each component construct. Control Flow Links from the end of each component construct are connected to the end of a different (single) Control Fork Join. The parallel construct runs its component constructs simultaneously, with the possibility of communication between them. Figure 5.13 shows an example of a parallel construct without communication (an example parallel construct with communication is shown in figure 5.8). Communication between branches of a parallel construct is dealt with in section 5.2.5.2 of this chapter. The GILT parallel construct is equivalent to the Occam parallel construct and obeys the laws of associativity, so that many nested parallel constructs are equivalent to a single large one. A parallel construct is invalid if any of its components may change the value of a variable which may be used in any of its other components. Restrictions on the connections to and from Channel Ports belonging to parts of parallel constructs also exist, and are discussed in the section on communications constructs (5.2.5.2). Parallel constructs are represented in the grammar by the non-terminal "PAR".

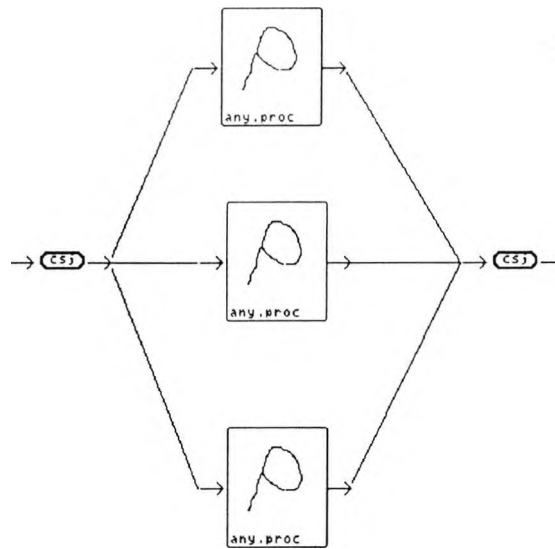


Figure 5.13 - An example parallel construct. Three instances of the Process Icon "any.proc" are shown, wired in parallel between the two Control Flow Fork Join Icons. No communication between the Process Icon instances is shown.

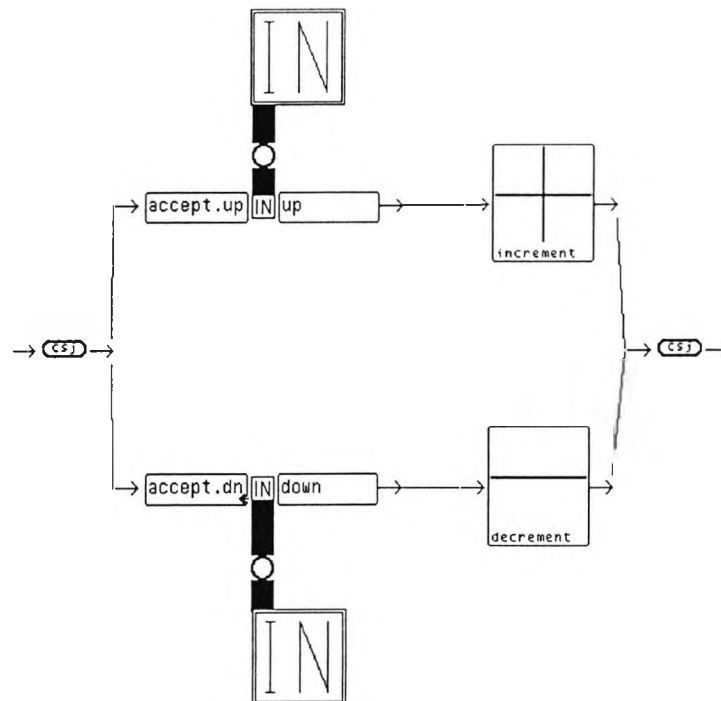


Figure 5.14 - An example alternative construct. Two guarded processes are shown. The guards contain textual shutoffs "accept.up" and "accept.dn" which may suspend branches of the alternative from operation. When a guard fires, a data value on a Channel Link (attached to the Channel Input Stub of each guard) is read into the appropriate integer variable "up" or "down" and execution of one of the processes "increment" or "decrement" begins.

5.2.5.1.5 Alternative construct

The alternative construct is similar in structure to a parallel construct, but each part of the construct has a Guard Icon connected between the starting Control Fork Join Icon of the alternative construct and the start of the construct in question. The construct is equivalent to the simple class of Occam alternative constructs discussed earlier and again is associative. Figure 5.14 shows an alternative construct. Alternative constructs are represented in the grammar by the non-terminal "ALT".

A Guard Icon in an alternative "fires" when an appropriate data item is available on the Channel Link connected to its Channel Input Port and when the boolean expression on its shutoff area is TRUE. If the shutoff area is empty of text, the guard behaves as if the shutoff condition is TRUE. During firing, the data item is read into the Guard Icon's variable (if defined), contained in the input variable area. Control then exits the Guard Icon via its Control Output Port into the appropriate construct. If both the shutoff area and the input variable area are empty so that the guard's variable is not defined, the guard behaves like the Occam guard TRUE, and provides a default action. Once one Guard Icon in an alternative construct has commenced firing, all others cease to execute and their control flow never exits.

5.2.5.1.6 Conditional construct

GILT's implementation of the conditional construct differs from Occam's in that multi-way conditionals are not supported. This is not important as multi-way constructs may be expressed using the existing notation. Two versions of the conditional are available, a visualisation of a conventional imperative "if" construct and a visualisation of a conventional imperative "if..else" construct. The if construct is equivalent to an Occam IF construct with two branches. The first branch contains a process to be executed if the condition is TRUE, while the second branch contains a SKIP process with a constant TRUE condition. This branch acts as a default and ensures that the conditional process always terminates. The if..else construct is equivalent to an Occam IF construct with a second TRUE branch containing a default process. Figure 5.15 shows an example if and an example if..else construct. Conditional constructs are represented in the grammar by the non-terminals "IF" and "IFELSE".

Control enters the conditional construct via the Control Flow Input Port of the construct's Conditional Icon. Depending on whether text in the Condition Icon's Condition Text Area is true or false, it exits either on the Control Flow Output Port or on the not Control Flow Output Port to the appropriate construct. If the construct terminates, control flow enters the Control Split Join Icon at the end of the construct and then exits.

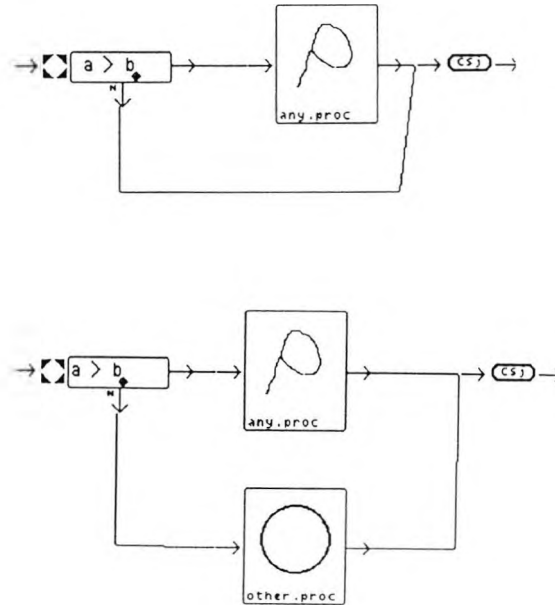


Figure 5.15 - A pair of example conditional constructs. An if construct (topmost) and an if..else construct are shown. Both constructs execute the process "any.proc" if the condition "a > b" is satisfied. If the condition is not satisfied the if construct passes control on to the Control Fork Join Icon. The if..else construct shown is similar, but executes the process "other" if the condition is not satisfied. If an executed process fails to terminate, control will not leave the construct.

5.2.5.1.7 While construct

The while construct is similar in form to the if construct but has a Control Flow Link which connects back to the Condition Icon's Control Flow Input Port. Figure 5.16 shows an example while construct. The process executes in similar fashion to the if construct described above, but control may only leave the construct after the conditional become FALSE. While constructs are represented in the grammar by the non-terminal "WHILE".

5.2.5.1.8 Other control flow constructs

Occam supports constructs which are not implemented at a visual level by GILT. These include replicators, CASE statements and complex guards in ALTs, amongst others. Their omission is not significant as enough constructs are included

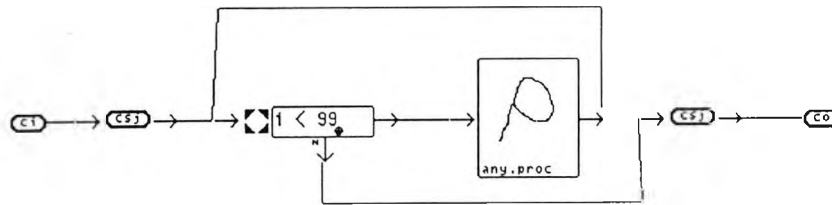


Figure 5.16 - An example while construct. The Process Icon instance "any.proc" is executed while the condition "i < 99" is satisfied.

in GILT to allow complex programs to be built. The inclusion of further constructs in GILT is discussed in the final chapter of the thesis, though it should be noted that such constructs may be used at a textual level in GILT programs.

5.2.5.2 Inter-process communication constructs

Inter-process communication in GILT relies on the connections made between Channel Connector Icons, Channel Input Stubs, Channel Output Stubs, Channel Input Ports and Channel Output Ports. A single Channel Link is allowed from a Channel Input Stub or Channel Output Port to a Channel Connector Icon. Similarly, a single Channel Link is allowed to a Channel Input Port or Channel Output Stub from a Channel Connector Icon. As many connections as are required are allowed to or from Channel Connector Icons, but a Channel Link may not be connected between two Channel Connector Icons.

Inter-process communication structures in GILT are modelled using a graph grammar which constrains the structures which may be evolved. In the grammar Channel Connector Icons, Channel Input Stubs, Channel Output Stubs, Channel Input Ports and Channel Output Ports are represented by terminal nodes. Channel Links between the components are represented by labelled arcs in the grammar. The Channel Input and Output Ports provide connection points for the attachment of communication graphs to a base graph. The approach has already been discussed in chapter four, and will not be further discussed in this section which is concerned with the semantics of the communication graphs, their relationship with Occam and the reasoning behind the choice of GILT's representation of inter-process communication.

Occam places relatively few restrictions on inter-process communication patterns in programs but, in essence, only processes which may be executing in parallel are allowed to communicate with each other. This restriction is enforced by the use of extra grammatical rules, as discussed in section 4.3.2.3. It is not possible to transfer such rules into the syntactic domain for GILT, so Occam's restriction on communication between sequential processes is restated in a suitably modified form for GILT :

"A sequence is rendered invalid if a Channel Connector Icon exists which has connections to the Channel Input Port of one functional icon and from the Channel Output Port of another functional icon".

GILT's grammar is therefore like Occam's insofar as it only specifies some facets of the inter-process communication process, but it may be considered more rigorous in other respects. It had been mentioned that the Channel Stubs in a Process Icon's definition diagram are analogous to the declaration of channel parameters in the header of an Occam procedure, but GILT is stricter than Occam and requires that the user specify the direction of communication for the channel by using an appropriate Channel Stub. Channel Input Stubs indicate that the declared channel is an incoming one, while Channel Output Stubs indicate that the declared channel is an outgoing one. The best way to imagine the system at work is to think about a version of Occam which requires such a rigorous definition. The earlier procedure definition for a buffer procedure in section 3.1.11 might then look like :

```
SC PROC buffer.1(INPUT CHAN OF INT in, OUTPUT CHAN OF
                  INT out)
  -- definition of single buffer process,
  -- using a more rigorous procedure header

  WHILE TRUE
    INT x :
    SEQ
      in ? x
      out ! x
  :
```

As Channel Links are directional, only outgoing Channel Links are allowed to connect from Channel Input Stubs. Conversely, only incoming Channel Links are allowed to connect to Channel Output Stubs. The system allows the directional flow of data along channels to be clearly visualised.

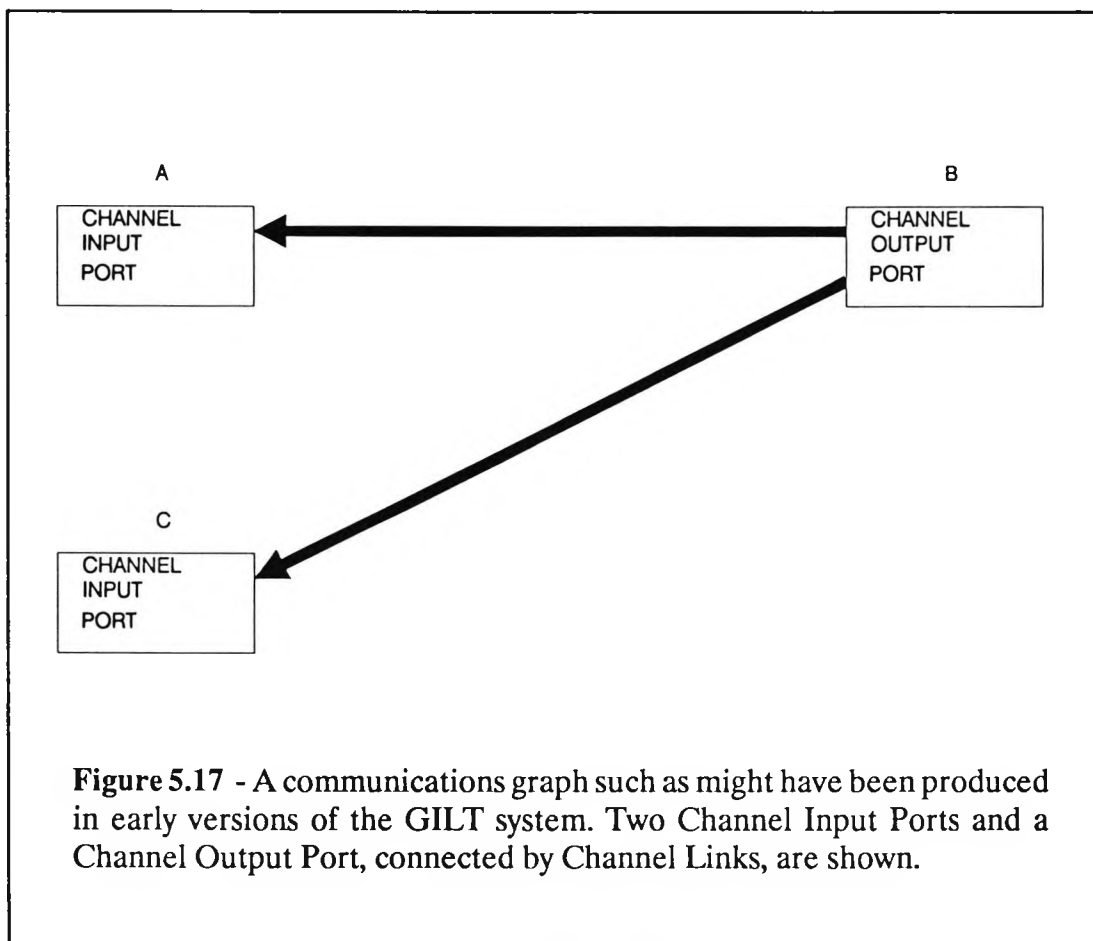
GILT's channel connection philosophy is that diagrams should be thought of without reference to the context in which instances of the Process Icon which they define are embedded. The notion of input and output stubs aids this process, supports stepwise refinement and code reuse, and aids the development of modular programs. In support of this notion, Channel Links in GILT do not have names. They are used purely for the specification of inter-process communication structures.

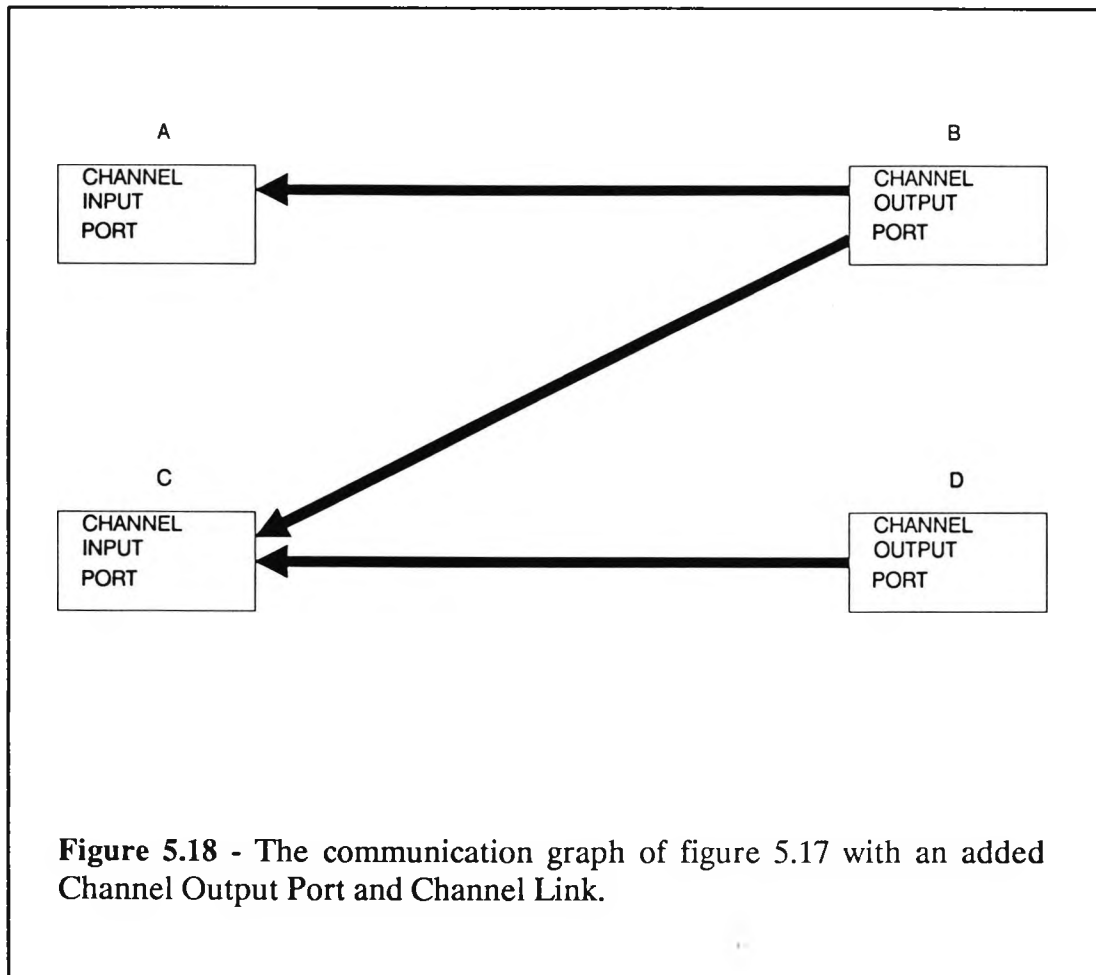
Channel Connector Icons were introduced to support this style of programming and to ensure that it is not possible to evolve a communication structure in GILT which is not part of the Occam computational model, or which has unclear meaning. Their introduction was made necessary by the use of control flow in GILT diagrams. Without Control Flow Links and with all processes assumed executing in parallel, specification of inter-process communication may be carried out by allowing point to point Channel Links between Channel Ports and Channel Stubs with no more than one connection per item. This system yields diagrams similar to bubble and arc diagrams.

Using a diagrammatic model with control flow adds a new requirement, namely that multiple ports should be able to be linked together by a common channel so that members of a sequential construct should be able to visually access the same channel.

An obvious approach used in early versions of GILT allowed channels to fork and join at the Channel Input and Output Ports of Process Icons. No Channel Connector Icons were used and communication was relatively unstructured.

Figure 5.17 shows a communication graph such as might have been produced by the early versions of the system. It contains two Channel Input Ports and a Channel Output Port interconnected by Channel Links. The Process Icon instances which

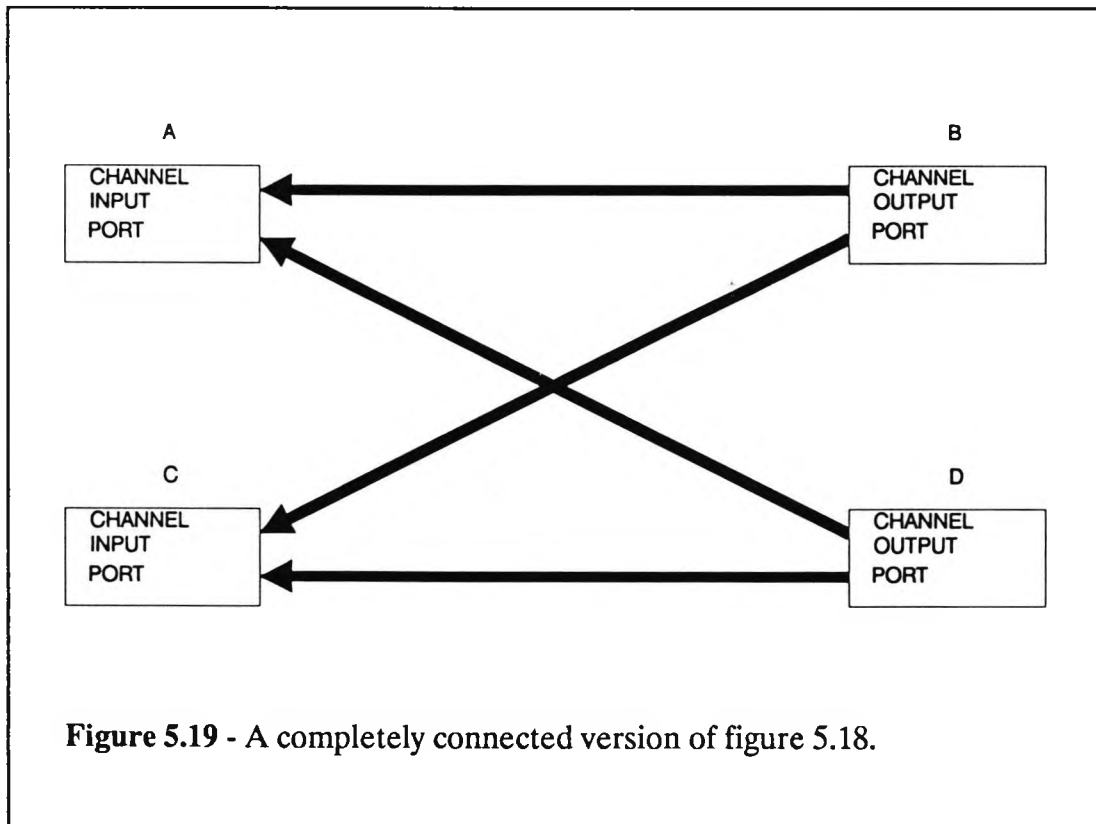




include the ports are not shown in the graph. Clearly, if the two input ports are part of Process Icons in a sequential control flow structure, then the graph is a potentially useful, legal one.

If a second Channel Output Port is added together with another Channel Link, the graph of figure 5.18 may be formed. Necessarily, ports B and D are part of a sequential construct, and any two Channel Links connected to the same Channel Port must form part of the same communications system. Thus port C reads from the same channel as does port A. However, the diagram visually implies that Channel Output Port D may only output to Channel Input Port C. Such a structure violates Occam's semantics and thus does not have a simple implementation (without channel multiplexing) in Occam. Any Occam implementation would allow all Channel Output Ports to output to a single channel which could be read by all Channel Input Ports. This implementation is not the one implied by the diagram. Similar arguments apply to structures containing Channel Input and Output Stubs.

Structures like the ones above could be clarified by ensuring that all Channel Output Ports in such graphs are connected to all Channel Input Ports. An example of such a structure is the fully connected version of figure 5.18 shown in figure 5.19. This scheme works well for graphs with only a few inputs and outputs. It is however



easy to see that the number of connections grows quickly with the number of Channel Output and Channel Input Ports. In fact, where "n" is the number of Channel Links, "i" is the number of input ports and "o" is the number of output ports, $n = io$.

Just four fully connected input and output stubs require sixteen connections, resulting in a mass of confusing connections, which is difficult to parse and to represent using a graph grammar.

The situation may be made considerably clearer with the addition of an extra graph component and a small set of rules. The extra component corresponds to a Channel Connector Icon. Only a single connection may be made outwards from a Channel Output Port or Channel Input Stub to a Channel Connector Icon. Similarly only a single connection may be made inwards to a Channel Output Stub or Channel Input Port from a Channel Connector Icon. As many connections as are required may be made to and from a Channel Connector Icon, but Channel Connector Icons may not be connected together. Every Channel Connector Icon must have at least one incoming Channel Link and one outgoing Channel Link.

Such a scheme results in flower-like communications structures with central Channel Connector Icons and radiating Channel Links. The Channel Links connect from the Channel Connector Icon to Channel Output Stubs or Channel Input Ports and to the Channel Connector from Channel Output Ports or Channel Input Stubs. A reduction in the number of connections needed to express a legal communications structure is obtained for realistic cases, with n reduced to $(i +$

o). Finally, the scheme ensures that users are aware that structures composed with Channel Links are language entities in their own right.

The rules may be formally written in the productions of a context free graph grammar. Figure 5.20 shows productions in the communications grammar based on the rules, which shares common terminal and non-terminal symbols with the base grammar. Figure 5.21 shows an equivalent graph to that shown in figure 5.19. Figure 5.22 shows a GILT diagram having a communications structure like the one represented by figure 5.21.

5.3 Textual process specifications

The nature and form of the graphical part of GILT programs has been discussed and it has been mentioned that the functionality of textual Process Icons is defined using Occam. This section is concerned with the syntax of the textual Process Icon definitions and their relationship with Occam.

The icon specific part of textual Process Icon instances is modelled by the terminal symbol "TEXTPROC" in GILT's grammar. Like instances of graphical Process Icons, textual Process Icon instances have ports arranged around the outside of the central area to provide connections for links. The overall external structure of the two forms of Process Icons is the same, even to the extent that they share many common productions in the grammar.

The Occam for textual Process Icon definitions is entered into pop up windows. Diagrammatic symbols arranged in panels around the outside of the windows declared the parameters for the textual Process Icon definition

Textual Process Icons, like Graphical Process Icons, may be procedural or non-procedural. Occam's syntax gives the structure of a procedure definition (equivalent to a procedural Process Icon) using the production :

```
proc.definition ::= proc.heading
                  process
```

The body of Occam's procedure definition is modelled using the "process" non-terminal, while the procedure header is modelled by the "proc.heading" non-terminal.

For a procedural Process Icon the panels around the outside of a text editing window define the equivalent of an Occam procedure header, while the central textual specification defines the body of the procedure. For non-procedural Process Icons the window is the equivalent of a textual macro definition. Figure 5.23 shows an example text window for a procedural Process Icon. To the top and bottom are two panels which are used to specify the text window's Channel Input and Output Stubs (and hence the Channel Input and Output Ports of the Process Icon which represents it). As in the earlier graphical process definitions, Channel Stubs define external channel connections for the specification. Each stub is

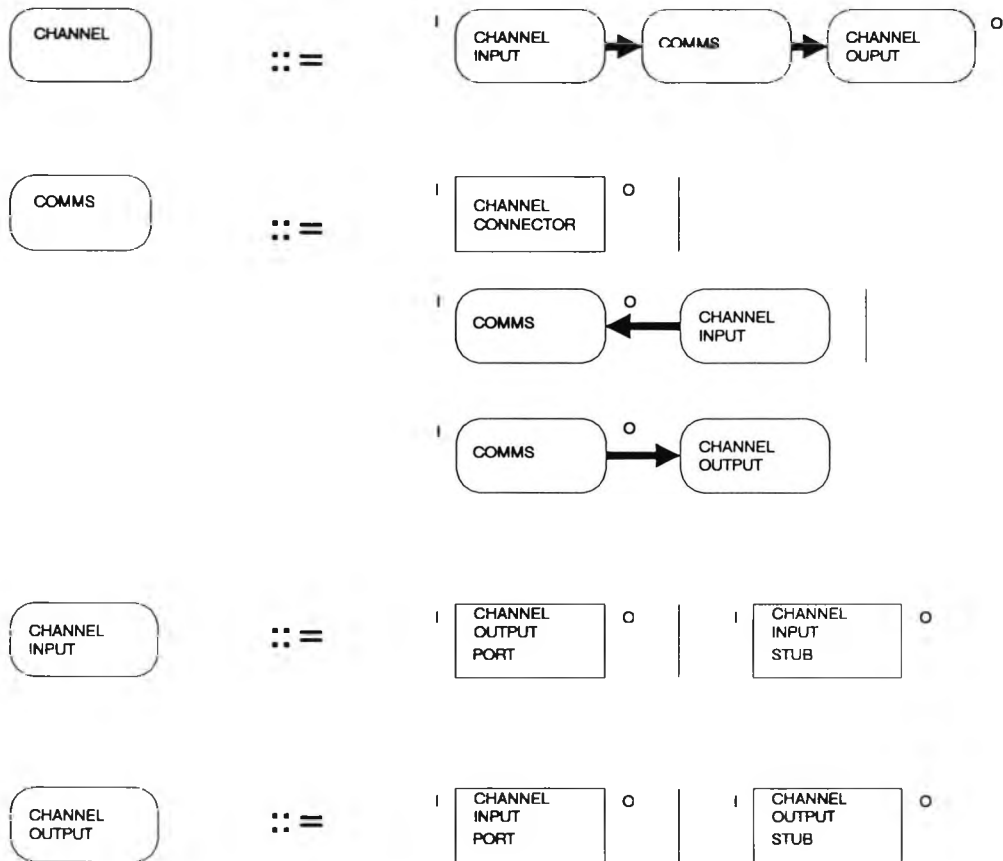


Figure 5.20 - Productions in a communications grammar which may be used to generate the graph of figure 5.21.

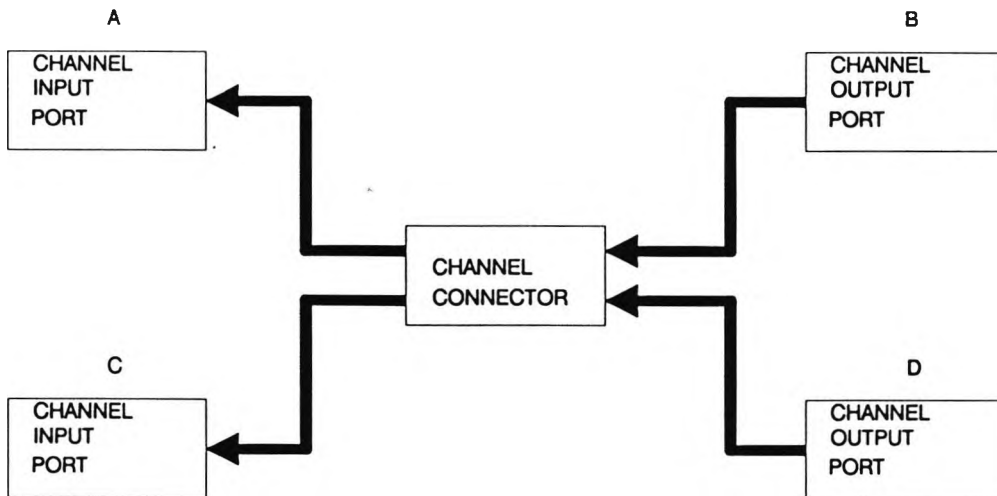


Figure 5.21 - An equivalent graph to that of figure 5.19 produced with the grammar of figure 5.20. Central in the graph is a terminal symbol representing a Channel Connector Icon. Channel connections radiate to Channel Output Ports and from Channel Input Ports.

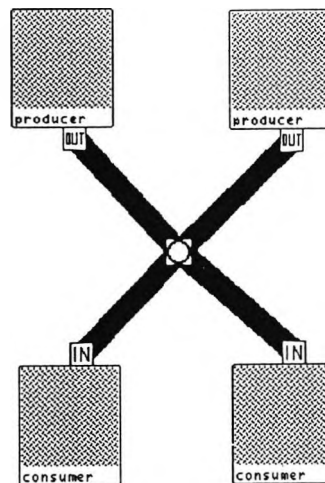


Figure 5.22 - A GILT diagram having a communications structure like the one in figure 5.21. No control flow information is shown in the diagram to clarify communications structure. Clearly however the two "producer" Process Icon instances and the two "consumer" Process Icon instances must be in two separate sequences (or sequential loops) running in parallel.

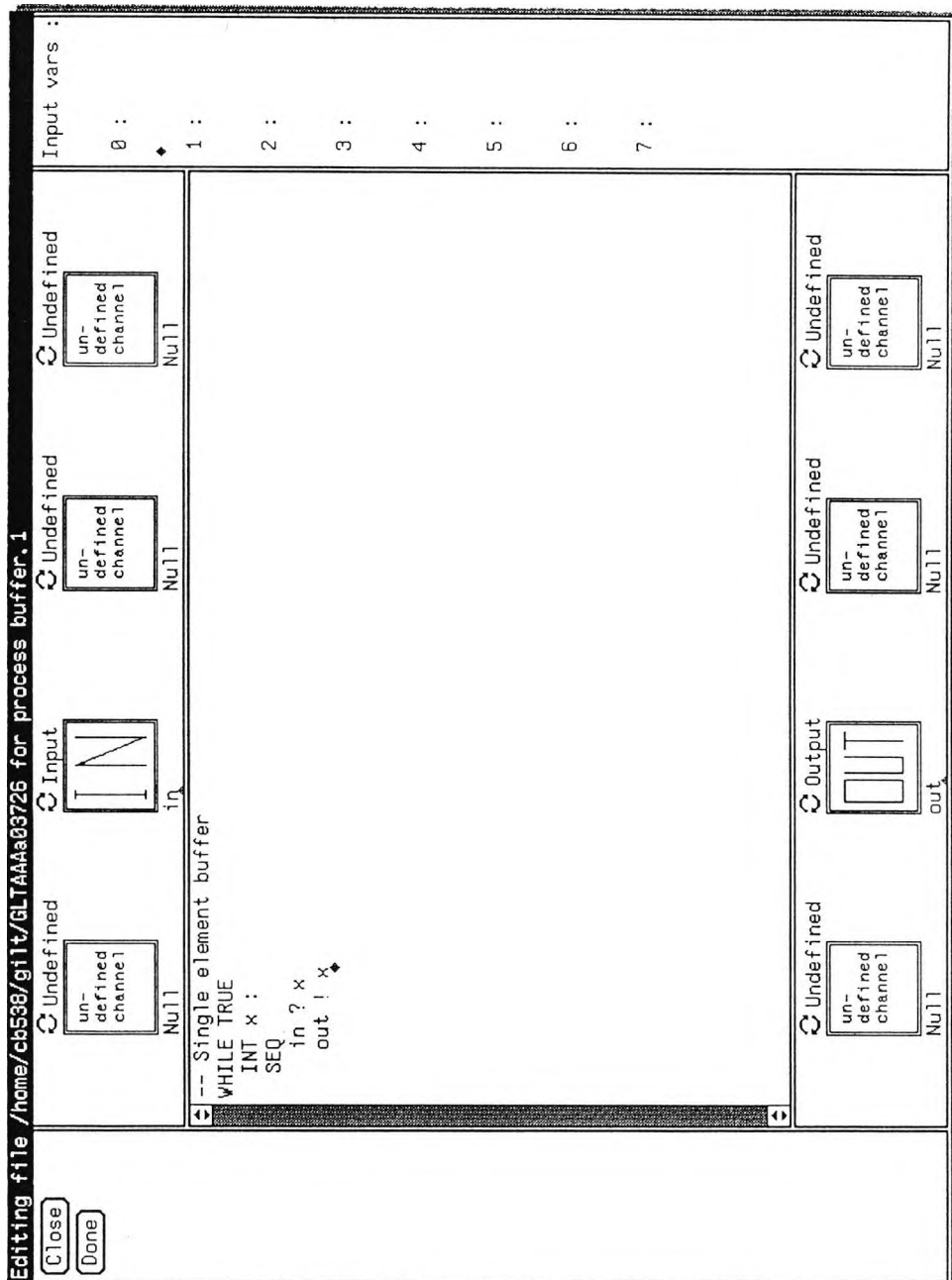


Figure 5.23 - An example text window showing a single element buffer process with one Channel Input Stub and one Channel Output Stub. To the top and bottom of the window are areas for the definition of Channel Input and Output Stubs. To the left is a control panel for the window. To the right is an area for the definition of non-channel parameters for the process equivalent to those defined by Declared Parameter Icons for graphical Process Icon definitions (none are defined). The central area holds Occam text, which implements a single buffer process.

named and may be in one of three states; undefined, input or output. The example of figure 5.23 has one Channel Input Stub and one Channel Output Stub defined, which are named "in" and "out" respectively.

If a stub is undefined, it does not appear on an instance of the Process Icon defined by the window. Undefined stubs simply serve to mark potential locations for Input and Output Stubs, while stubs in either of the two other states define the Channel Inputs and Channel Outputs of a text window. Channel Input Stubs appear as Channel Input Ports on the Process Icon defined by the text window, while Channel Output Stubs appear as Channel Output Ports. A similar algorithm to that described earlier in respect of graphical Process Icons (section 5.2.4) relates the positions of stubs and their representative ports. Channel Stubs in textual windows are named, unlike those used in graphical process definitions. A stub's name may be used in Occam's input and output statements and provides an interface between the graphical stubs and the textual process code. GILT's Channel Links currently support only integer channel protocols, so integers only may be input to or output from the stubs using Occam's "?" and "!" operators.

The text that may be entered in the Text Editing Area is defined to have Occam's "process" non-terminal as its root symbol, and thus is allowed access to nearly all the facilities of Occam with the exception of the process to processor mapping statements and other related functions. A full definition of the components of the "process" non-terminal may be found amongst the syntax of Occam given in appendix 1. Within a textual process, processes may be run in parallel, local procedures defined, replicators used and arrays of channels using any of Occam's protocols created. GILT is thus a unique current visual programming system in that it allows parallel programming to be performed at a textual level. Most visual programming systems restrict textual programming to sequential code, ignoring the possibility that text may provide a clearer expression of some parallel constructs than may be provided by graphics.

5.4 Programming with GILT

This section provides a step by step analysis of the features that GILT provides for concurrent programming by considering an example application (a processor farm) and a common parallel programming structure (a circular buffer). To illustrate GILT's high level programming features the processor farm is examined, while for low level features the circular buffer process is used. Both examples present Occam code fragments which are equivalent to the GILT diagrams implementing the examples in order to illustrate how GILT's visual display of parallelism, communication and control sequencing is helpful in visualising software designs, and to enable readers to make a connection between the constructs of Occam and those of GILT. The two examples do not use all of GILT's facilities, but are sufficient to allow the essence of GILT programming to be demonstrated. It should be noted that it is difficult to describe hierarchical GILT diagrams without using the GILT editing system to examine them, as the process of viewing a program in GILT is far more interactive and visual than in

conventional textual languages. The examples presented here are also necessarily small in dimensions.

5.4.1 A processor farm in GILT

Processor farms have been used extensively for solving so called "embarrassingly parallel" problems. Such problems are computationally expensive, but may be divided into a large number of component tasks. Good examples of embarrassingly parallel problems amenable to processor farm implementation are the calculation of Mandelbrot sets, ray-tracing algorithms and many image processing applications. Because such problems may be divided into many component pieces, almost any implementation strategy will take advantage of the available parallelism, but processor farms provide a very simple general method which provides almost linear speedup with the number of available processors in very many cases.

In a processor farm a "worker" process is run on all but one of the available processors in a parallel machine. An additional process, the farmer process, farms out "tasks" or "task packets" to each of the worker processes. When a worker finishes its task, the results are transmitted back to the farmer process for assimilation into the complete solution to the problem being processed and a new task is transmitted to the worker concerned. The processing continues until all of the available tasks have been transmitted, at which time the farmer collects the remaining results from its workers and emits a termination signal. Variants on the scheme exist, for example with each worker storing a task, so that "gaps" in computation caused by worker to farmer latency do not exist.

Processor farms are commonly implemented in Occam for connected "pipelines" of Transputers like the one used for the buffering example of chapter three (figure 3.3). The Occam for the main part of such a processor farm is shown in figure 5.24.

The instance of the procedure "farmer" sends out work to the farm in packets, while "no.of.workers" worker processes labour on the data. Packets flow out via the arrays of channels "to.farm" with processed work returning on the channel array "from.farm". Each "worker" process removes the work it requires from the "to.farm" channel or, if it does not require any more, forwards the work to the remainder of the pipeline. The process at the end of the pipeline "end.worker" does no forwarding and so is a special worker with only one incoming and one outgoing channel and no packet forwarding.

"Bubble and arc" diagrams of processor farms are often included in papers on the subject to clarify the structure of a farm. Such a diagram is not reproduced here in order to illustrate how GILT's representation of the processor farm is clearer than the corresponding textual (Occam) one. Figure 5.25 shows two views of a three worker processor farm (equivalent to the Occam of figure 5.24 with no.of.workers = 2). The top view shows a GILT diagram for the farm with control flow "on" (displayed), the bottom with control flow "off". The diagram, and the following sub-diagrams (for the component Process Icons), are suitable for a

```
[no.of.workers + 1]CHAN OF INT to.farm :  
[no.of.workers + 1]CHAN OF INT from.farm :  
PAR  
  farmer(to.farm[0], from.farm[0])  
  PAR i = 0 FOR no.of.workers  
    worker(to.farm[i],to.farm[i+1],from.farm[i+1],  
          from.farm[i])  
  end.worker(to.farm[no.of.workers],  
            from.farm[no.of.workers])
```

Figure 5.24 - Occam for the main part of a processor farm. The existence of the procedures "farmer" "worker" and "end.worker" is assumed. No configuration information is included.

distributed implementation of the GILT compiler's diagram parsing algorithm, discussed in chapters seven and eight. All of the Process Icons in the example are procedural.

In GILT's version of the farm, parallelism is immediately obvious. As all of the Process Icons are part of the same parallel construct, the control flow information (shown by the Control Flow Links and Control Flow Ports in the top diagram of figure 5.25) is to some extent redundant, and the diagram is better viewed without control flow, as in the bottom diagram of figure 5.25. The pipeline structure of the application may be clearly seen, with the Channel Links between the Process Icons explicit and visual. It should be noted that, when coding applications in GILT, it is difficult to misconnect channels because the communications structure is so obvious. The erroneous connection of channels is far easier in Occam where it may occur by misordering the passed parameters for a procedure instance. GILT's communications structure can equally well be displayed using default Process Icon images created by the system, as shown in figure 5.26, rather than the specialised ones used in figure 5.25. Specialised icons are useful for tidying up an application with "visual comments", for the depiction of the functionality of commonly used processes, like buffers, or for indicating what a process does with its channel data.

The internal detail of the component processes making up the farm may also be expressed by GILT diagrams. Figures 5.27, 5.28 and 5.29 show respectively the internal detail of the "farmer", "worker", and "end.worker" Process Icons, while figure 5.30, 5.31 and 5.32 give the Occam equivalents. Control flow is not shown in any of the diagrams, as all of the component Process Icons are in parallel (or in the case of the "end.worker" Process Icon, one strictly sequential Process Icon) and there would be little point. It should be noted that the GILT program editor allows the toggling of control flow in the diagrams, so that it may be viewed or not as is required (or even rapidly "toggled" for checking purposes). Again, GILT's

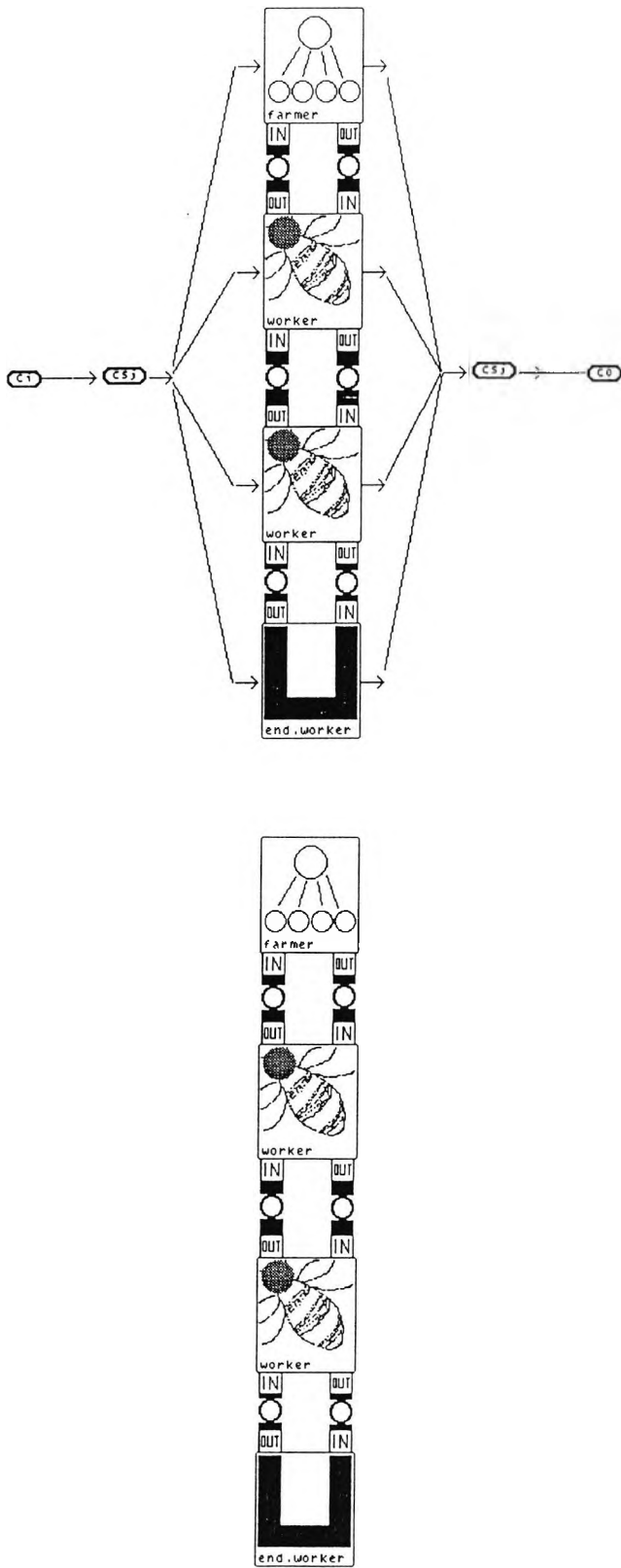


Figure 5.25 - Two "top-level" views of a processor farm in GILT.

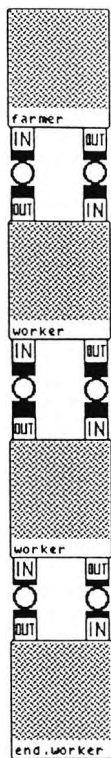


Figure 5.26 - A view of the farm without specialised icons.

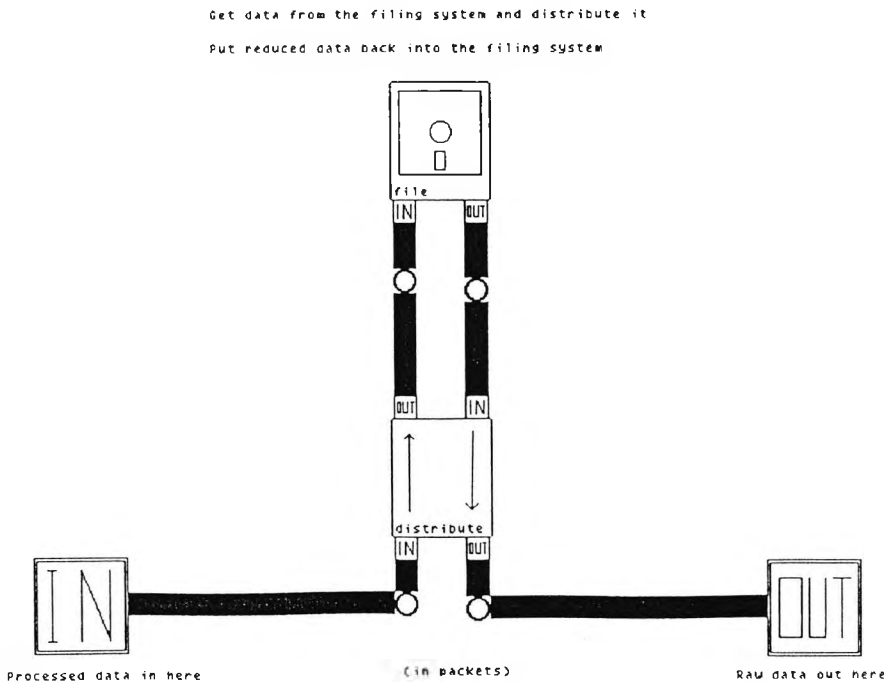


Figure 5.27 - Definition diagram for a "farmer" Process Icon.

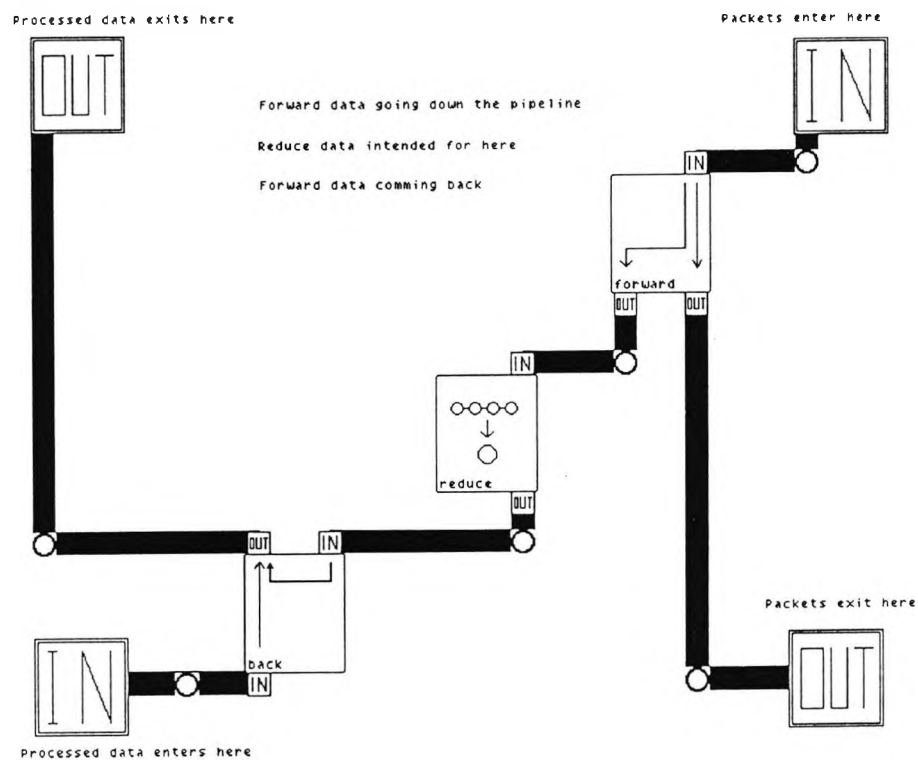
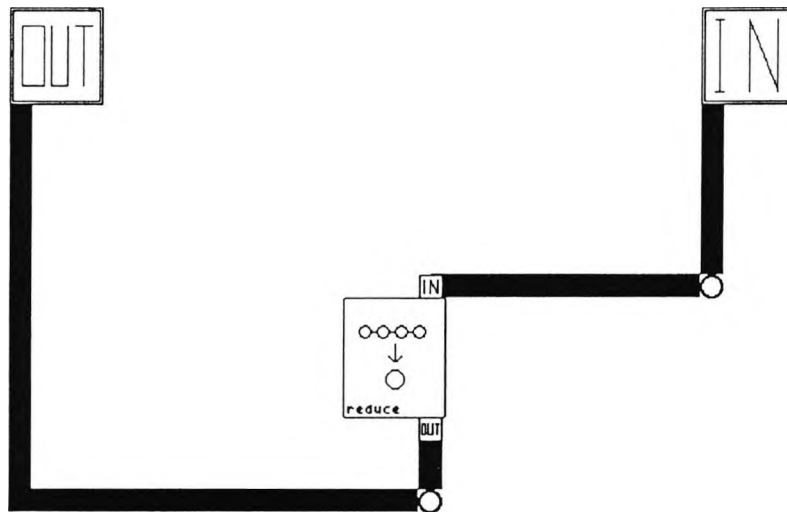


Figure 5.28 - Definition diagram for a "worker" Process Icon.



The end of the pipeline needs no packet forwarding

Figure 5.29 - Definition diagram for a "end.worker" Process Icon.

```
PROC farmer(results.in, data.out)
  CHAN OF INT to.file, from.file :
  PAR
    file(to.file, from.file)
    distribute(from.file, data.out, results.in,
              to.file)
  :
```

Figure 5.30 - Occam for a "farmer" process.

```
PROC worker(work.in, work.out, results.in,
results.out)
  CHAN OF INT to.reduce, from.reduce :
  PAR
    forward(work.in, work.out, to.reduce)
    reduce(to.reduce, from.reduce)
    back(from.reduce, results.in, results.out)
  :
```

Figure 5.31 - Occam for a "worker" process.

```
PROC end.worker(work.in, results.out)
  SEQ
    reduce(work.in, results.out)
  :
```

Figure 5.32 - Occam for an "end.worker" process.

graphical representations aid the clear expression of the internal workings of the of the Process Icon, in the style of dataflow diagrams, which is common for overviews of parallel programs shown in the literature and for the higher level of abstraction in GILT programs.

The "farmer" Process Icon has two component Process Icon instances, both part of a parallel construct. The first, "file", interacts with a mass storage system and removes tasks from a data file and places the results from completed tasks back into a separate file. "Distribute" passes out task packets to the "worker" Process Icons further down the pipeline, and collects processed data for transmission back to the "file" Process Icon.

"Worker" Process Icons have three component Process Icon instances; "forward", "reduce" and "back". "Forward" accepts task packets from the upper rightmost Channel Input Stub, passing those intended for other workers to the lower rightmost Channel Output Stub. Packets for the "reduce" Process Icon are forwarded to it as appropriate. "Reduce" emits its packets of processed data to "back", which combines them into the stream of packets heading up the pipeline towards "farmer". Of note are the "forward" and "back" Process Icon images. The use of iconic representation allows the direction of the data flow in the process represented by the icons to be shown explicitly, so that the splitting action of

"forward" Process Icon and the combining action of the "back" icons may be clearly shown. No such technique can be used in textual programming languages.

The "end.worker" Process Icon is a specialised worker process which does no forwarding of task or processed data packets.

In all three diagrams the channel connections between processes are explicit and visual with the "IN" and "OUT" Channel Ports and Stubs clearly showing the direction of data flow in the system.

Further definition diagrams could be produced to describe the component Process Icons of the "farmer", "worker" and "end.worker" Process Icons, thus dividing up the functional specification of the program into even smaller modules. This course is not pursued here as the internal details of the modules in the farm would start to become application specific rather than general. Nonetheless, the example serves to illustrate GILT's facilities for programming at a high level of abstraction, much like dataflow design methods. The next example illustrates how GILT's lower level facilities may be used to aid programming at lower levels of abstraction.

5.4.2 A circular buffer in GILT

Buffers are commonly used in parallel programming because they allow a processes to run decoupled from their data sources or sinks. An example application of a buffer is to be found in the previous processor farm, where buffering might be required to decouple the actions of the packet forwarding "forward" Process Icon instance and the "reduce" Process Icon instance.

The circular buffer is a useful structure which allows a buffer of generalised size to be created in an efficient manner. The buffer's data structure is an array of fixed size with two associated pointers "base" and "top" indicating the next object to be taken from the buffer's array and the next free "slot" in the array respectively. With each "put" or "get" operation on the buffer the pointers are incremented using modular arithmetic so that they "wrap" around the top of the array (hence the term "circular"). Obviously an implementation of the buffering process must not allow a "get" from an empty buffer, or a "put" to a full one, so a check on the number of items in the buffer is kept separately from the pointers. Figure 5.33 shows an Occam implementation of the circular buffering process, which consists of two main processes, a circular buffer and a single element buffer.

The code of figure 5.33 is necessarily complex because input and output operations on a single channel may not be contained in guards of different ALT processes, as this would violate Occam's semantics. The code implements a buffer of size $(S + 1)$, with one buffered element held in the process "buffer.1". To read from the circular buffer the "buffer.1" process first indicates that it wishes to read by synchronising down the channel "request", then actually reads via the channel "reply". External input is performed along the channel "put" and output along "get". The interaction between the circular buffer and the single buffer takes the form of a double rendezvous, common in Occam programs.

```
PROC buffer.1(CHAN OF INT request, in, out)
  INT temp :
  VAL INT any IS 0 :
  WHILE TRUE
    SEQ
      request ? any
      in ? temp
      out ! temp
  :

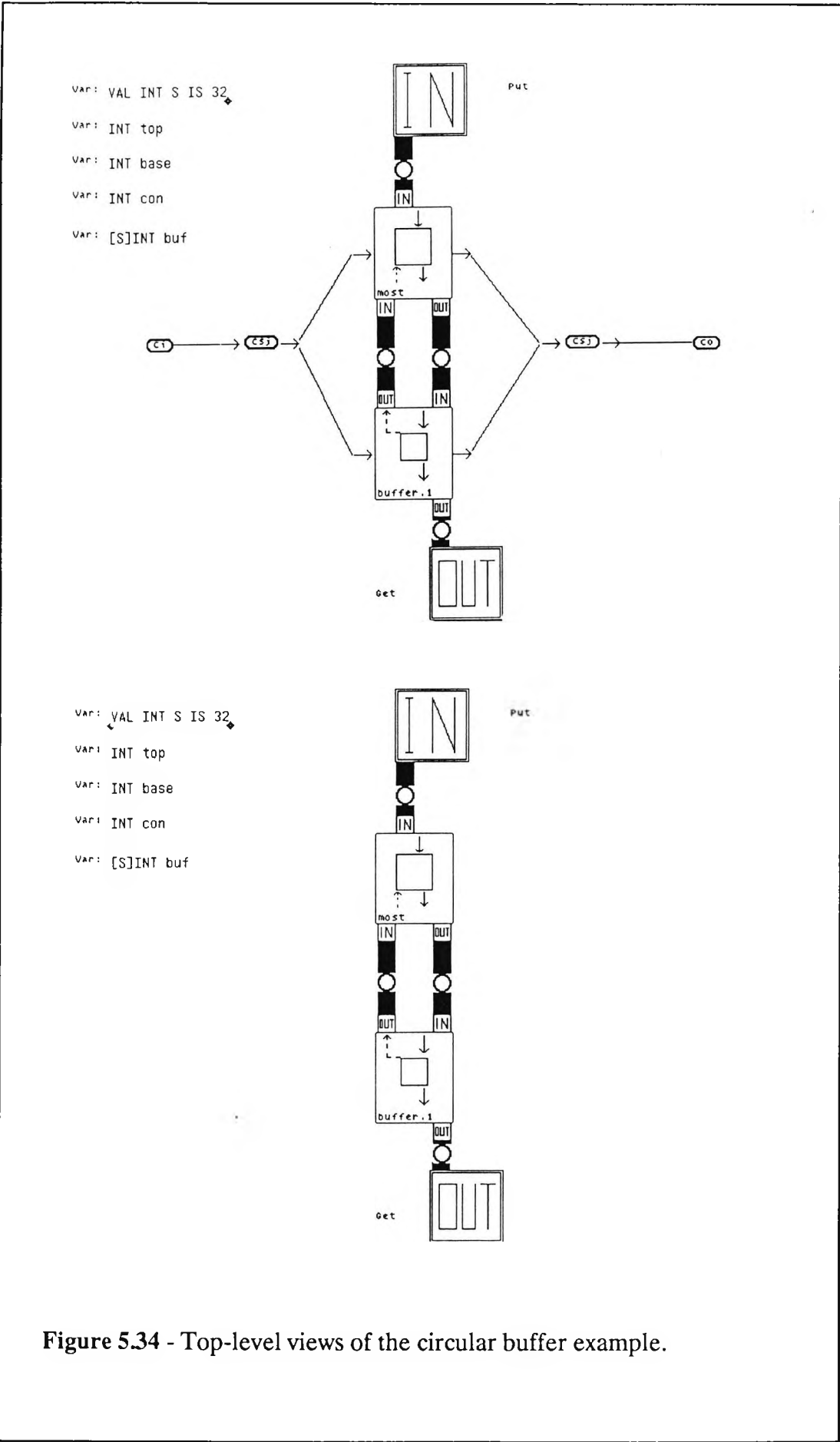
PROC circbuf(CHAN OF INT put, get)
  VAL INT S IS 32 :      -- The buffer size
  INT top, base, con : -- next object out, next in
                        -- number of objects in ..
  [S]INT buf :          -- buffer array
  CHAN OF INT request, reply :

  PAR
    -- "most"
    SEQ
      -- init.vars
      con := 0
      top := 0
      base := 0
      -- docircle
      WHILE TRUE
        -- putnget
        INT any :
          ALT
            con < S & put ? buf[top]
            -- put
            SEQ
              con := con + 1
              top := (top + 1) REM S
            con > 0 & request ? any
            -- get
            SEQ
              out ! buf[base]
              con := con - 1
              base := (base + 1) REM S
          buffer.1(request, reply, get)
  :
```

Figure 5.33 - Occam code implementing the circular buffer algorithm.

In GILT, a circular buffer may be implemented as a multilevel diagram, which is discussed in the following paragraphs. The Occam code of figure 5.33 contains comments with the names of the various Process Icons mentioned in the following discussion, so that the Occam code may be related to the diagram, and vice-versa.

Figure 5.34 shows a "top-level" view of the circular buffer in GILT. The top view shows the Control Flow Links in the definition diagram, which executes the two Process Icon instances named "most" and "buffer.1" in parallel. The lower view, as for the earlier processor farm example, shows a view without control flow information. As expected, "most" contains the majority of the code for the buffer. The Channel Links between the two Processes may clearly be seen, as can the structure of the buffer as a main process ("most") with a subsidiary single element buffer process ("buffer.1"). The various local variables for the buffering process are declared by the Variable Declaration Icons on the left hand side of the diagram, and are equivalent to those in the earlier Occam version of the buffer. The channel used for rendezvous is marked by a dotted arrow in the raster of each of the two Process Icon instances "most" and "buffer.1". External connections of the buffer process are seen as the Channel Input Stub to the top of figure 5.34 and the Channel Output Stub at the bottom of the same figure. Figure 5.35 shows a text editing window for the Process Icon "buffer.1", which is equivalent to the "buffer.1" procedure definition of figure 5.34 and is not further discussed. Figure 5.36 shows a definition diagram for the Process Icon "most", which is non-procedural. Two Process Icon instances (in a sequence) are contained in the definition diagram, "init.vars", and "do.circle". "Init.vars" is a textual non-procedural Process Icon which zeroises the variables "con", "top" and "base" declared in figure 5.34. The text for "init.vars" is trivial and is not reproduced. The Process Icon "do.circle", also non-procedural, is graphically defined (instead of textually) and executes a while-true loop, as shown in its definition diagram (Figure 5.37). Both the definition diagrams shown in figures 5.36 and 5.37 have similar communications structures, passing channels down to lower levels of abstraction, and both are non-procedural. The lowest level definition diagram is that of the non-procedural Process Icon "putnget" which is defined by the definition diagram shown in figure 5.38. The mixed control flow and channel based visualisation allowed by GILT allows a clear expression of the alternative structure of figure 5.38, showing clearly how the two Process Icon instances "put" and "get" are executed dependent on data items received on the Guard Icon's connected Channel Links. The fact that "get" outputs on receiving a request is clearly shown by the diagram. Both "put" and "get" are non-procedural textual Process Icons, for which text editing windows are shown in figures 5.39 and 5.40 respectively.



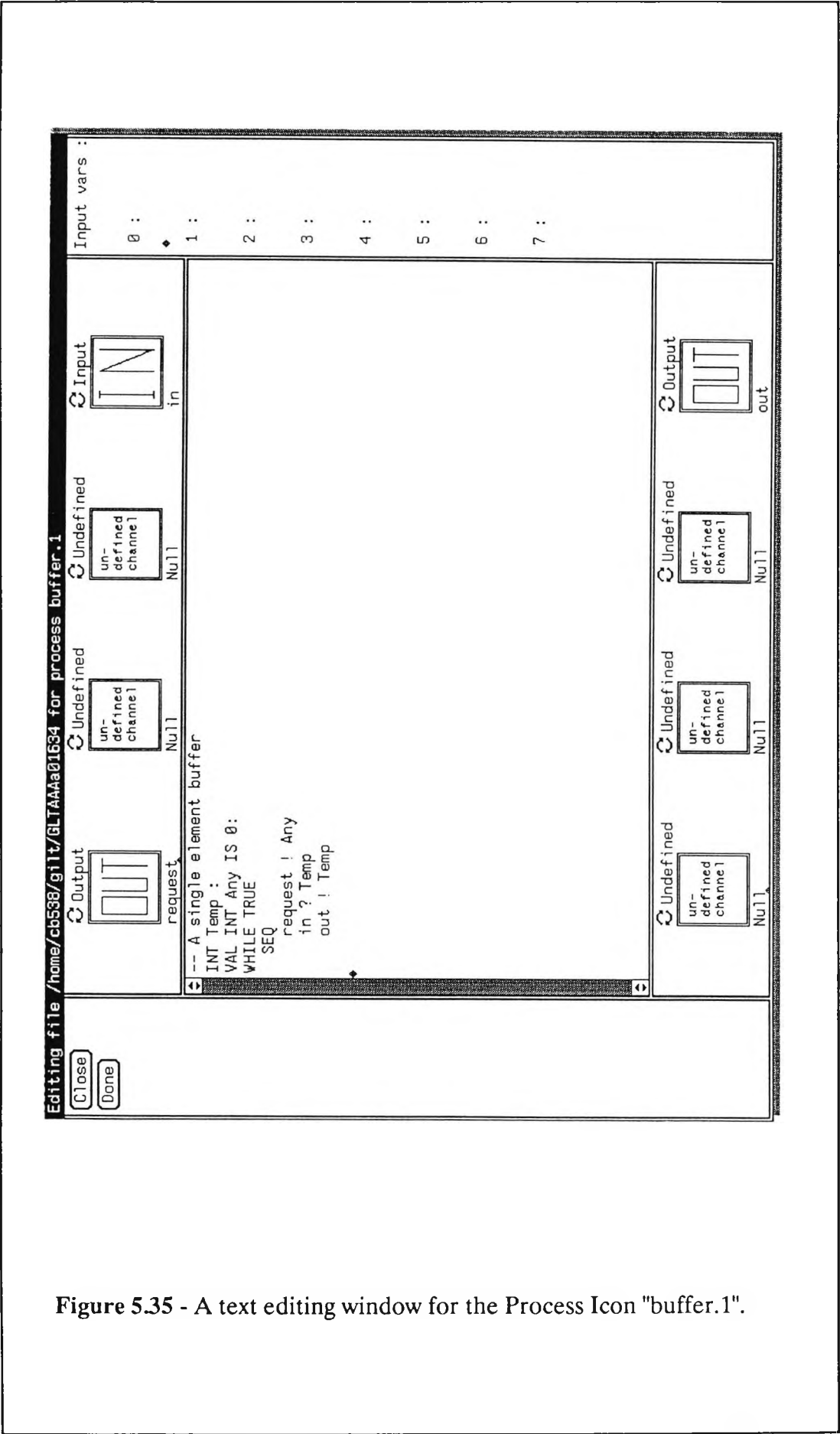


Figure 5.35 - A text editing window for the Process Icon "buffer.1".

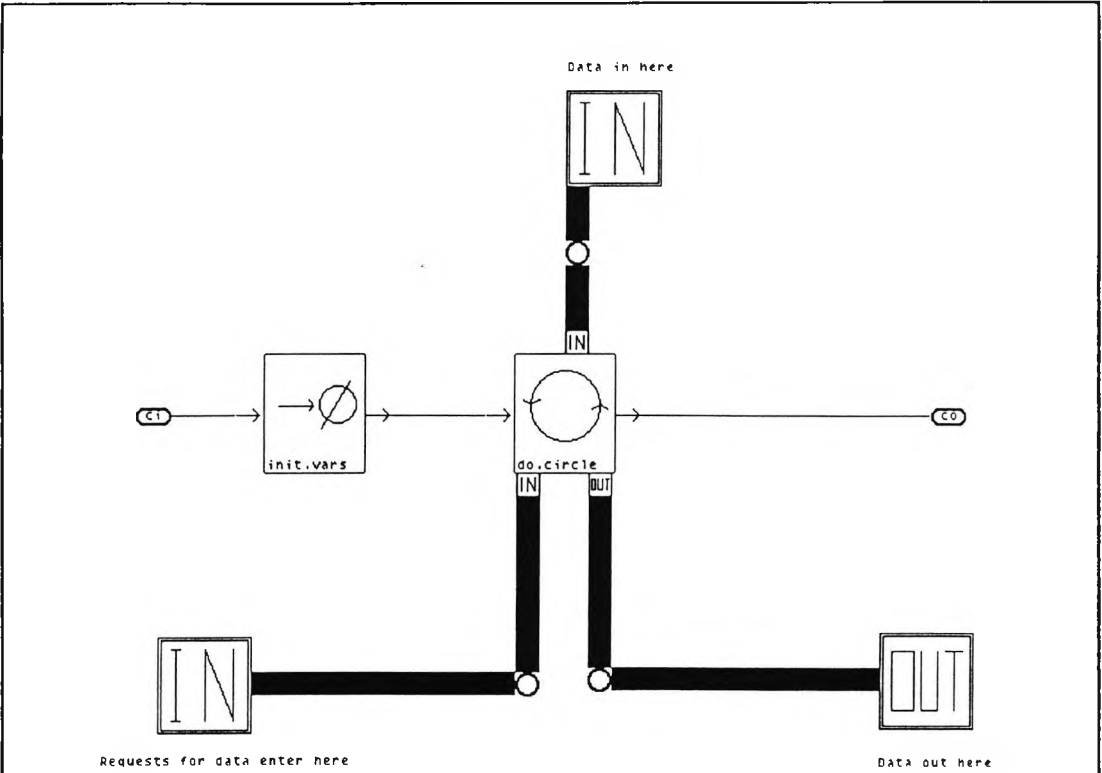


Figure 5.36 - The definition diagram for the Process Icon "most".

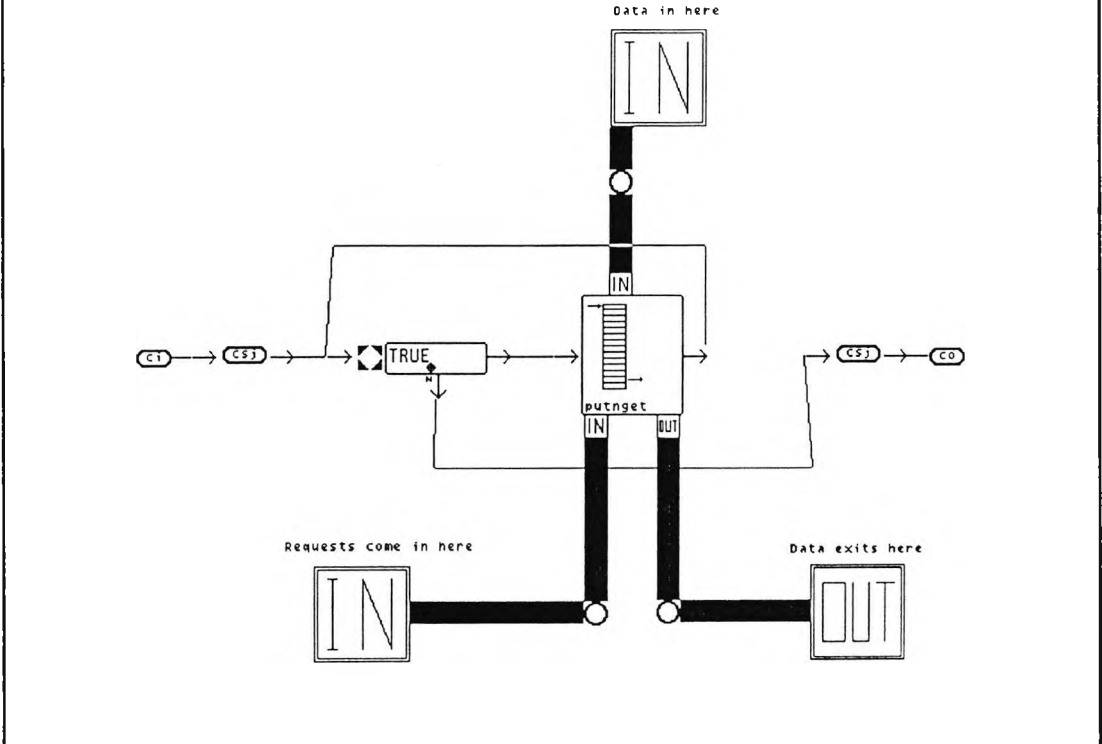


Figure 5.37 - The definition diagram for the Process Icon "do.circle".

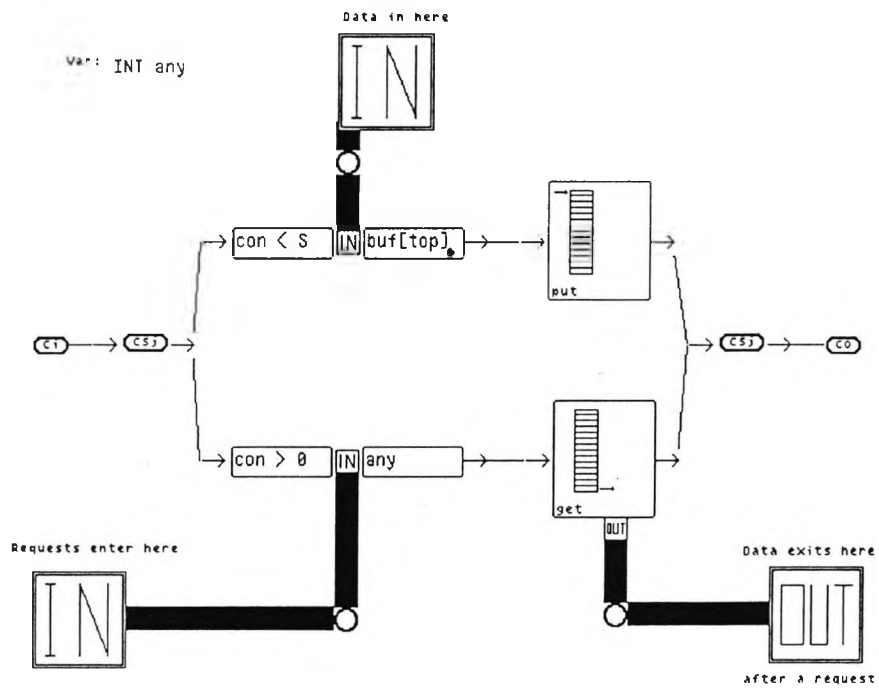


Figure 5.38 - The definition diagram for the Process Icon "putnget".

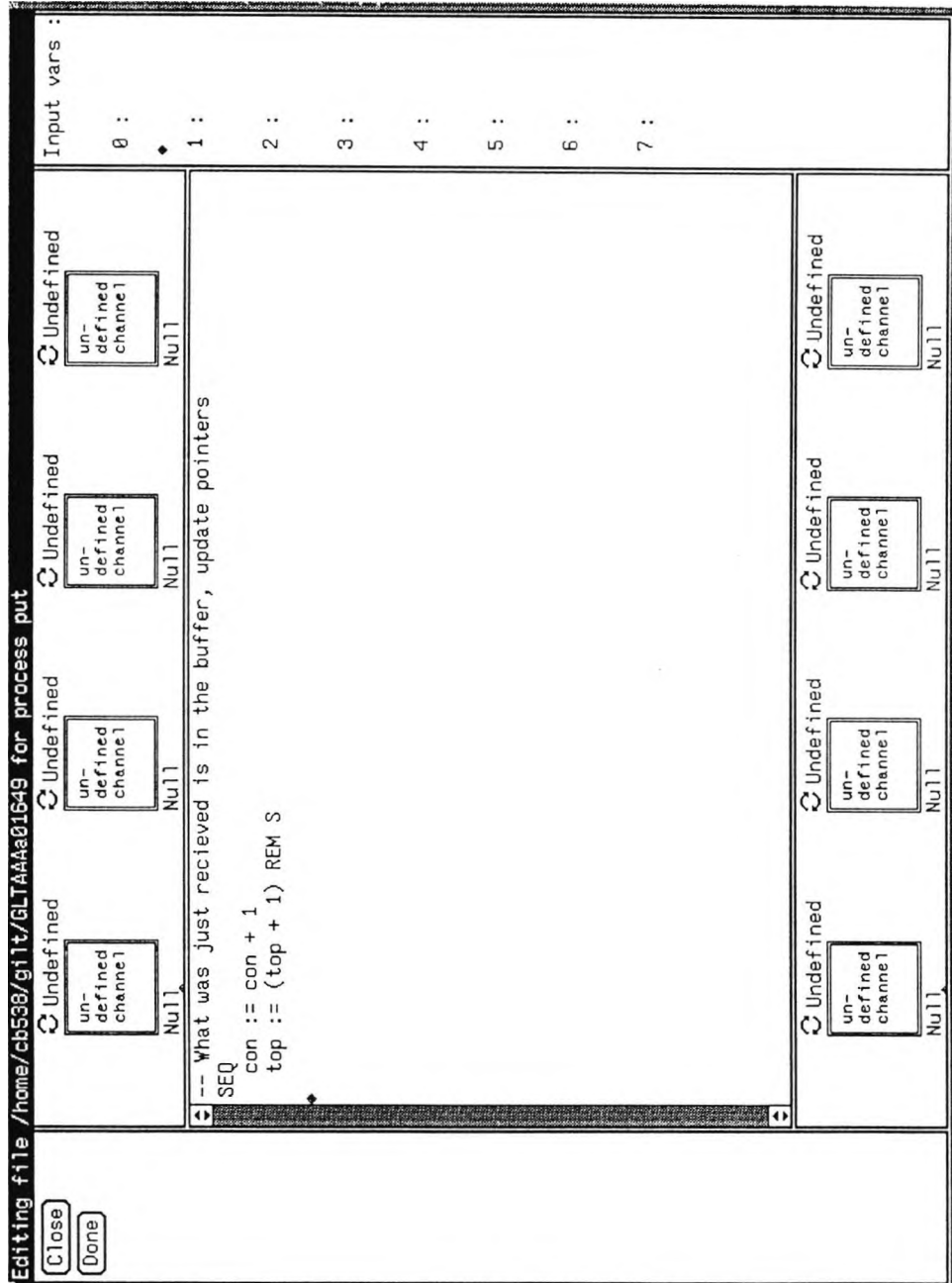


Figure 5.39 - A text window for the Process Icon "put".

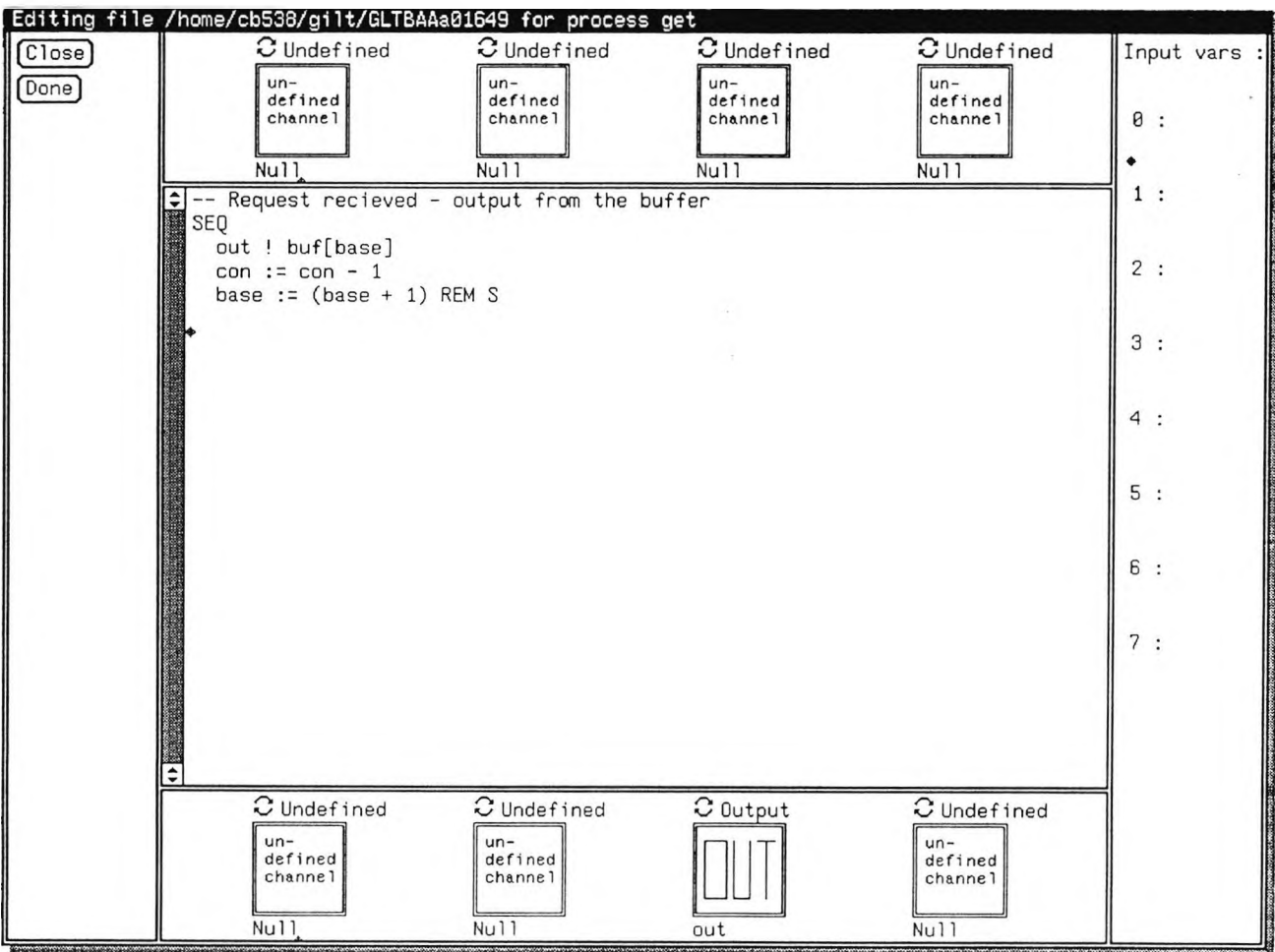


Figure 5.40 - A text window for the Process Icon "get".

An editor for GILT diagrams

6.0 Introduction

Many editors for textual languages have been produced. Such editors are frequently far simpler than corresponding visual language editors due to the one dimensional nature of text. The existence of a well accepted basic symbol set for textual languages (ASCII) has allowed editors with a high level of generality to be produced so that the implementation of a new textual language does not usually require the implementation of a new editing system.

Attempts have been made at the implementation of "universal" editors for graphical (visual) programming languages, for example (Hekmatapour and Woodman, 1987; Gottler, 1989), but the level of generality of application for such systems is low, caused in part by the diversity of diagramming methods used in current visual programming languages and the lack of a basic symbol set. The development of a completely general system may be precluded by the lack of orthogonality between approaches to visual programming. Certainly no existing editing system would be suitable for editing GILT programs due to GILT's heavy use of representative icons and its reliance on a mixed textual-graphical paradigm.

6.1 Editing system overview

GILT's editing system was heavily influenced by the decision to implement a runtime system which supported the traditional edit..compile..edit code development cycle. In the cycle, programs are developed using a program editor, compiled, and error messages used to refer back to erroneous features of the source code. Most previous visual programming language editors have supported the development of programs using syntax directed editing facilities, which only allow the input of correct (or potentially correct) program fragments. This path was not chosen because of the previously mentioned desire to support a traditional code development cycle, and because differences between the grammar produced for GILT and conventional textual grammars required the development of a new checking and compilation system. The system was considered better approached in a more modular form than was offered by a fully syntax directed editor.

The editor does however impose some restrictions on the diagrams which are input. These restrictions are termed "lightweight restrictions" because they impose little computational load on the editing system and take the form of restrictions preventing, for example, the connection of Channel Links to Control Flow Ports. All "heavyweight" checking is performed by the compilation system.

6.2 Systems for the implementation of the editor

Several windowing systems were considered for the implementation of the diagram editor. After an initial trial (partial) implementation of the diagram editing system on a Whitechappel Mg-1 workstation a number of criteria were identified for an implementation environment :

- a) The environment should support applications consisting of multiple adjustable windows able to display text and graphics within the same area so that the mixed textual graphical nature of GILT diagrams could be emphasised and parts of the program not continuously used could be opened, closed, moved or resized at will.
- b) A large "widget toolkit" of user interface components should be readily available. The components in the toolkit should be easy to use, yet flexible enough to implement most (if not all) of GILT's features, so that as little software as possible concerned with low level user interface operations, and not directly with the material of the thesis, had to be written.
- c) The system should be in common use to ensure software support and inter-machine portability.
- d) A callback or object oriented style of user interface programming should be supported to allow freedom from intimate handling of user input.
- e) The system should run on a machine capable of hosting a Transputer development system.

Examination of option (e) reduced the hardware platform to either a Sun or a PC based system, with available windowing systems at the time being X-Windows or Sunview (on a Sun workstation) or Digital Research GEM (on a PC). After some consideration GEM was removed from the list as it did not provide sufficient facilities, in particular with regard to (c) and (d). The final selection of Sunview over X-Windows was made on Sunview's wide usage within the City University, its large widget toolkit (including useful features like text editing windows), and the fact that it was supported by a machine capable of hosting a suitable Transputer development system (a Sun 4/110). X-Windows was rejected due to the low level of support available within the University and its smaller widget toolkit (at the beginning of the implementation).

6.3 The GILT program editing system

The program editing system for GILT may be divided into two broad functional units, the graphics editor and the text editor. In the following discussion, the graphics editor will be discussed first, followed by the text editor. Each discussion consists of a functional description of the facilities provided, followed by some notes on the implementation approach taken.

Both the systems were written in C (Kernighan and Richie, 1988) and use Sunview library routines. Throughout existent code was reused or modified so that a fully functional editor could be produced quickly.

6.3.1 The graphics editor

GILT's graphics editor provides support for programming at the graphical levels of abstraction provided for in GILT. The editor allows the definition of Process Icons and the creation of definition diagrams containing them. A flexible icon editing system is provided for drawing Process Icon images (forming part of Process Icon representations), while a Process Icon library stores all available Process Icons for easy access.

For the purposes of extensibility and maintenance the editor was written in a highly modular fashion. Specialised routines handle the drawing of language components, which are internally represented using C data structures for speed.

The workstation mouse is used for the entry of diagram components, the editing of Process Icon images and for other functions. A control panel associated with each functional area or window controls the action of the mouse within the area. The system is designed to conform to the style of interaction used in Sun system software (Sun, 1989) to ensure uniformity between the editor and other programs.

As the editing system requires a large number of different functions, a number of controls are used to define the behaviour of the mouse when editing.

Heavy use is made of pop-up windows which perform tasks like querying potentially destructive actions, for example the deletion of diagram components.

The editor performs only very simple checking on the diagrams which are input, disallowing meaningless concepts like the connection of Control Flow Links to Channel Ports or Stubs and obviously erroneous communications structures. Implementation of a full syntax directed editor has not been attempted for the reasons advanced earlier. The simple grammar of the communications structures has allowed the editor to be implemented so that only legal communications structures (or parts of them) can be input. No attempt is made to ensure that the context holding the communications structures is correct (for example, it is

perfectly permissible to connect a Channel Link between two Channel Ports on Process Icon instances which form part of a sequence). Earlier discussions have shown that this type of structure cannot be outlawed using syntactic methods as it requires extra syntactic rules. The addition of such rules to the part of the editor which checks communications structures would not however be difficult.

GILT's graphics editor display consists of a main window containing two large functional areas (an icon editing area and a diagram editing area) with a pop-up window (the Process Icon library or browser) which gives access to Process Icons. The text editing system and GILT's compilation system also use pop-up windows.

Figure 6.1 shows the main window for the system, while figure 6.2 shows the smaller, pop-up, Process Icon library.

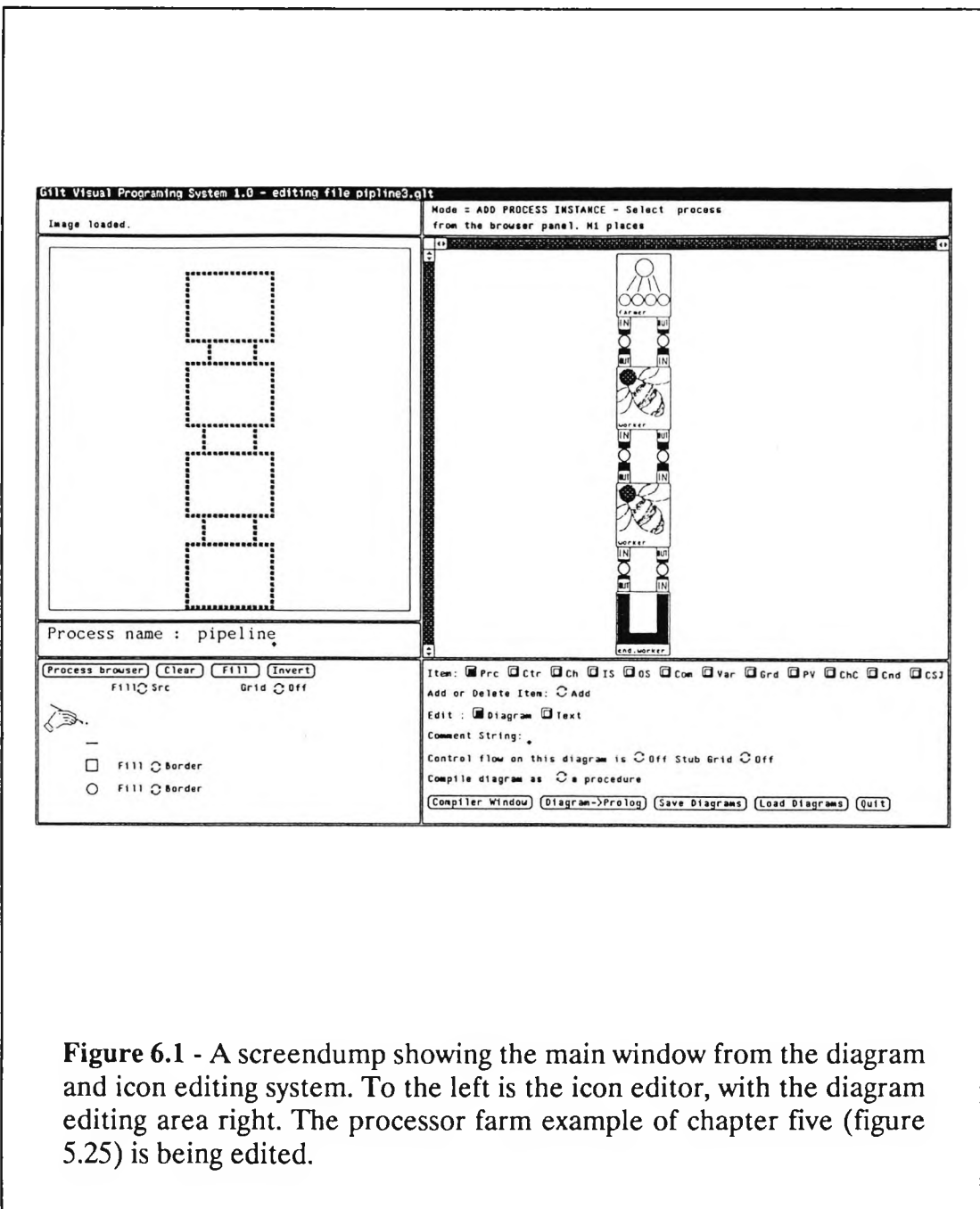


Figure 6.1 - A screendump showing the main window from the diagram and icon editing system. To the left is the icon editor, with the diagram editing area right. The processor farm example of chapter five (figure 5.25) is being edited.

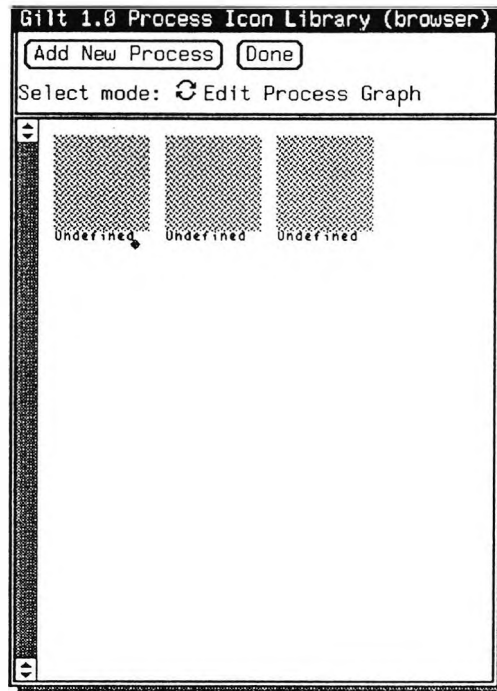


Figure 6.2 - The pop-up Process Icon library showing three newly created (and unedited) Process Icons ready for selection. The panel is accessed via the button marked "Process Browser" in the main window.

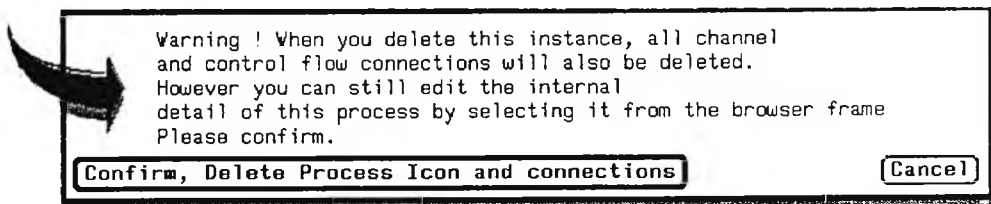


Figure 6.3 - A pop-up window requesting confirmation of a delete action.

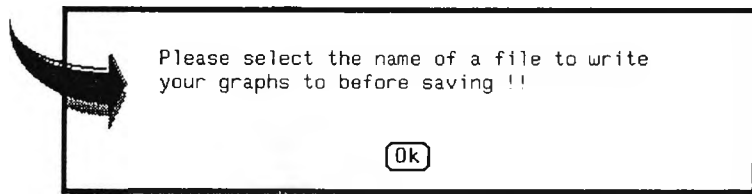


Figure 6.4 - A pop-up window giving assistance on the use of the system.

Actions which have potentially dangerous effects on a user's GILT program (such as the deletion of diagram components) are protected by pop-up windows which allow cancellation of the actions. Figure 6.3 shows an example of such a window. Pop-up windows are also used to display advisory information, for example, when a user tries to save a program without selecting a filename for it. Figure 6.4 shows an example of this type of window.

The three main components of the graphics editor are the Process Icon editor, the diagram editor and the Process Icon library or browser.

6.3.1.1 The Process Icon editor

The Process Icon editor, based on Sun's Iconedit program (Sun, 1989), is a flexible system for editing the representative images used for Process Icons and providing a definition of their name. The editor's display consists of four main sub areas (from top to bottom) :

The message panel

At the top of the area is the message panel. The message panel is used for system and help messages relevant to the actions taking place in the other areas of the Process Icon editor.

The drawing area

The next area down from the message area is the drawing area. This area is used for interaction with a Process Icon's image.

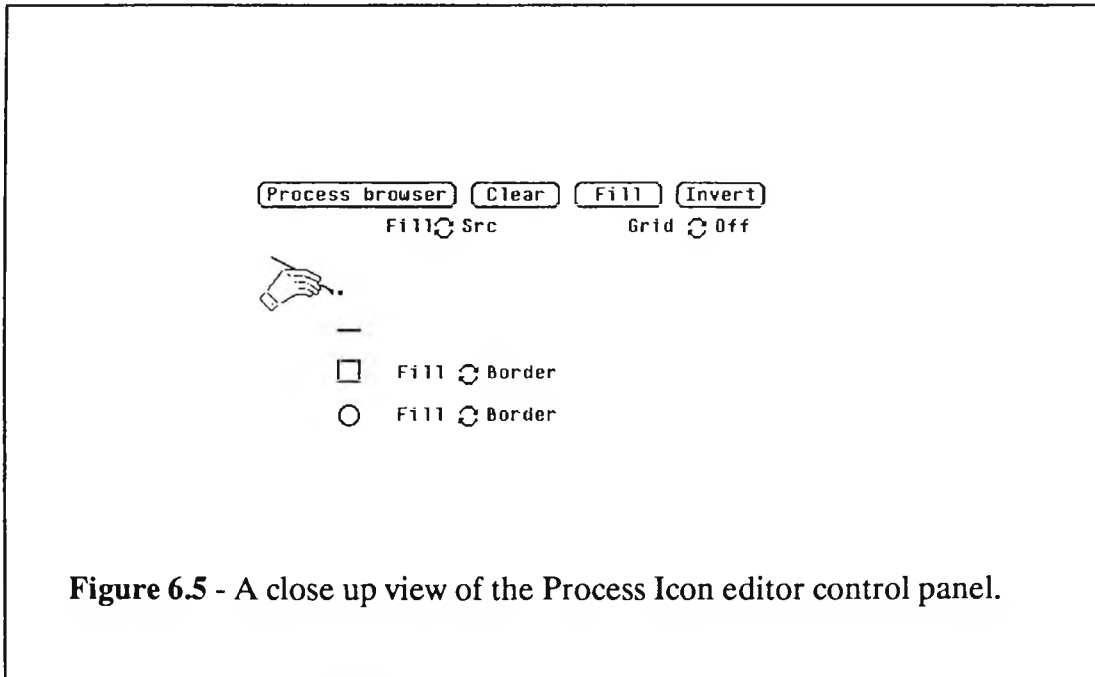
The Process Icon name area

The Process Icon name area is used to define a name for the Process Icon and is below the drawing area.

The control panel

The control panel displays the options currently available for the creation of images in the drawing area, determines the actions that the mouse takes in the drawing area, and provides a button which displays the Process Icon browser. The control panel is below the Process Icon name area.

Before creating an image for a Process Icon or editing the Process Icon's name, a Process Icon must have been created and selected for editing using the Process Icon browser, described in a later section. Once a Process Icon has been selected, the mouse may be used to edit the icon's image within the drawing area. The left button is used to draw, the middle button to erase. The rightmost mouse button may be used to undo the previous operation in the drawing area. As drawing proceeds, an enlarged version of the icon's image appears within the drawing area. A smaller "life-size" version may be found in the Process Icon browser's pop-up



window. The images of any instances of the Process Icon within diagrams are updated as editing takes place.

The Process Icon's textual name is entered into the process name area by positioning the mouse within the Process Name area and typing. Standard keys for delete and cursor control operations may be used.

6.3.1.1.1 Process Icon editor controls

The control panel contains a number of items which may be used to control image entry in the drawing area and one item which is used to display the library of available Process Icons stored in the browser. Some items are buttons which initiate commands, while others allow selection from a range of options.

Most items also have a menu which can be selected using the right mouse button.

A close up view of the control panel is shown in figure 6.5, with each item in the control panel described below, from left to right and downwards :

Process browser (button)

Pressing this button brings up a pop-up window (the Process Icon library or browser) containing the images and names of the Process Icon definitions known to the editing system. Using the window icons may be created, destroyed, selected for editing or placed in the diagram editing area. The Process Icon browser is described in a further section of this chapter.

Clear (button)

The clear button clears the currently selected icon's image contained in the drawing area.

Fill (button)

The fill button fills the currently selected icon's image with the current rectangular fill pattern, discussed in the section on the paintbrush item.

Invert (button)

The invert button flips the icon's image so that black becomes white and white becomes black.

Fill (cyclical choice item)

This item controls the "operation" used when filling parts of the drawing area. It contains a number of options showing logical functions (e.g. "AND", "OR") which combine the data displayed in the drawing area with that used in filing.

Grid (cyclical choice item)

The grid item displays a rectangular grid, useful for creating regular images, over the drawing area.

Paintbrush (vertical choice item)

The paintbrush item allows selection of four painting modes. Each mode is shown by a small icon, with a "pointing hand" indicating the "current choice". The modes are described from the top downwards :

"Dot" In the first mode, dot, pixels at the mouse's location in the drawing area are painted when the left or middle buttons are pressed.

"Line" The second mode, line, allows lines to be drawn. In this mode, a line is drawn from the position at which the left or middle mouse button is pressed down to the position at which it is released. While a button is held down "rubberbanding" is used to give feedback.

"Rectangle" The third mode allows rectangles to be drawn. Rectangles are defined using the same method as for lines (starting and finishing co-ordinates) and a rubberbanded rectangle is shown during drawing.

The "fill" item to the right of the rectangle selection item indicates the current rectangle fill pattern. Any rectangles drawn will be filled using this pattern, which is also used by the "fill" button.

"Circle" The fourth (and final) mode allows circles to be drawn using a similar technique to that used for rectangles.

A "fill" item like that for the "Rectangle" option is also provided for circle drawing.

6.3.1.2 The Process Icon library (browser)

The Process Icon library (Process Icon browser) is used to display, create, delete and select Process Icon definitions for editing or Process Icon instances for placement in diagrams. Figure 6.2 shows the Process Icon library. The browser's window consists of two main areas. At the top is a control panel containing three control items. Items in the control panel are used to create new processes, hide the window and to control the action of selection operations (made using the mouse) on the icons held in the lower part of the window. The lower part of the window consists of a scrollable panel which displays all of the Process Icons stored in the editing system.

6.3.1.2.1 Process Icon Library controls

The control panel has three control items, each of which has a different function. The different items are shown in figure 6.6, which is a close up view of the control panel :

Add new process (button)

The add new process button creates a new Process Icon in the Process Icon library. The Process Icon's image is initially grey and it has the name 'undefined'. Icons created in this way may be selected for editing (using the icon and diagram editing areas or a text editing window).

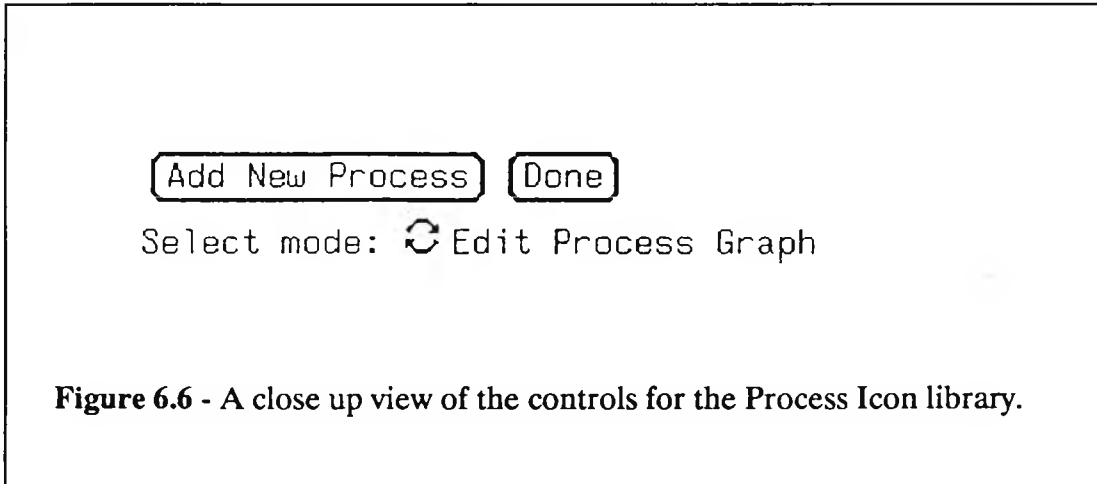
Done (button)

The done button closes the Process Icon library window, hiding it from view. It may be brought into view again using the "Process browser" button in the icon editor control area.

Select mode (cyclical choice item)

The select mode item affects the action taken when a Process Icon in the browser is selected by depressing the left mouse button while the mouse is over the icon. Three modes are available :

a) Edit process graph This is the initial mode. When a Process Icon is selected in this mode, the icon become the icon which is being edited. Its image appears for modification in the icon editing area, and its definition diagram (if it has one) appears in the diagram editing area. Changes to the icon's image, name, or definition diagram may then be made as required.



b) Place process When a Process Icon is selected in this mode, an instance of it may be placed in the diagram editing area by moving the mouse into the diagram area and depressing the left mouse button at an appropriate location (once selected, the icon "sticks" to the mouse pointer). Process Icons selected in the browser are shown enclosed by a rectangular box. Icons remain selected until the left mouse button is depressed over a different Process Icon.

c) Delete Process This mode is used for the removal of unwanted Process Icons from the browser. Once an icon has been selected, a small confirmation window appears, allowing the user to proceed, and delete the icon, or cancel the operation.

6.3.1.3 The diagram editor

The diagram editor works in a similar manner to the Process Icon browser and Process Icon editor, but is used for the creation of the diagrams which define the functionality of Process Icons (definition diagrams). The area has three main parts; the "message panel", the "diagram editing area" and the "control panel", shown in figure 6.7. The message panel (at the top of the diagram editing area) is used for the display of system messages and for help on editor functions. The diagram editing area (middle) is used for the display, creation and modification of diagrams using the mouse and keyboard. The functionality of the mouse buttons within the area is controlled by the items in the control panel (bottom). Items in the control panel are used for system functions, for example the loading and saving of graphs.

6.3.1.3.1 Diagram editor controls

The control panel for the diagram editor contains twelve items. Three items are concerned with the functionality of the mouse in the diagram editing area. These items will be referred to as the "mode controls".

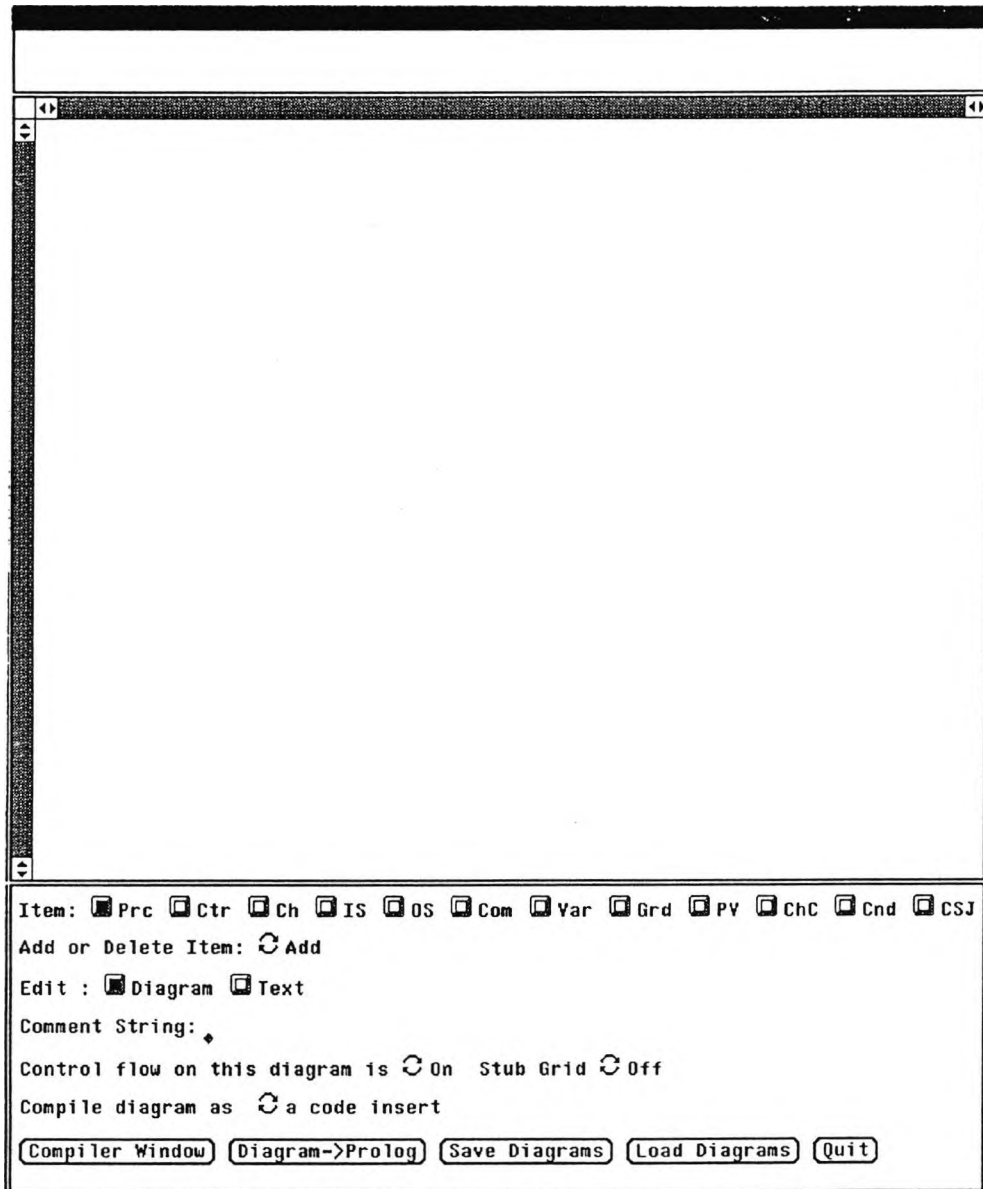


Figure 6.7 - An enlarged view of the diagram editor parts of the editor display. The diagram editor is shown in its initial state. No Process Icon is being edited.

Three "toggle items" are concerned with the way in which diagrams are displayed in the diagram editing window and with the nature of the diagram being edited in the display area (procedural or non-procedural).

Five "system items" perform functions like exiting the editing system, displaying the compiler window and loading or saving diagrams.

One item allows the entry of very long text strings which define the contents of comment items. Strings may be entered into this item (the "comment string item") and placed at any location on the screen using the mouse.

6.3.1.3.1 Mode controls and editing functions

The mouse is used in the diagram editing area for the entry of links between icons and for the placement, movement, deletion and selection of diagram components. The keyboard is used to type into text areas selected with the mouse.

The left mouse button is used for selection operations such as the addition and deletion of functional icons, the connection of links between them and for "zooming in" to see the internal detail (diagrams or text) of Process Icons. It is also used for the selection of text areas, which form part of icons, so that the contents of the areas may be edited using the keyboard.

The middle mouse button is used for moving components. Components in the diagram editing area may be moved at any time by depressing the middle mouse button over the component, moving the mouse to the desired location, and releasing the button.

The right mouse button is used for cancelling operations. Operations in progress (such as the entry of control flow links) may be cancelled by pressing the right mouse button.

The three mode controls, the "Item" choice item, the "Add or Delete Item" choice item, and the "Edit ." item, are used to determine the functionality of the left button of the mouse when the mouse pointer is in the diagram editing area.

Item (horizontal choice item)

The "Item" mode control item determines which of twelve diagram components are to be worked with in the diagram editing area. A mnemonic is associated with each choice in the item, only one of which may be selected at any time. The mnemonics used and their relationship to GILT's diagram components are shown below :

Prc	Process Icon
Ctr	Control flow links

Ch	Channel links
IS	Channel input stub icons
OS	Channel output stub icons
Com	Comment icons
Var	Variable definition icons
Grd	Guard Icons
PV	Passed or Declared variable icons. (The same item is used for both these diagram components as they are structurally the same, and may be distinguished by their textual contents).
ChC	Channel connector icons
Cnd	Conditional Icons
CSJ	Control split-join icons

No option for interaction with Control Input Stubs and Control Output Stubs are provided. As these components must be present in each diagram they are automatically created by the program editor whenever a new Process Icon is created using the Process Icon library's "Add New Process" button.

Add or Delete Item (cyclical choice item)

The "Add or Delete Item" mode control item allows the addition of new components to a diagram ("add" option) or the deletion of existing ones ("delete" option).

Edit : (cyclical choice item)

The "Edit :" mode item enables the diagram editing functions controlled by the other two mode items and controls the action of the left mouse button when it is over a Process Icon instance. When the item is in "Diagram" mode the diagram displayed in the diagram editing area may be edited, or icons entered to reveal their internal definition diagrams. In "Text" mode no diagram editing may be performed, but if the left mouse button is depressed over a Process Icon instance a text entry window may be created for the Process Icon's definition and text entered as required.

The message panel above the diagram editing area continuously displays help on using the editor for whatever mode is selected.

6.3.1.3.1.1 Adding diagram components to a definition diagram

New components are added to a diagram with the "Add or Delete Item" item in "Add" mode and the "Edit :" item in "Diagram" mode.

For icons ("Item" modes "Prc", "IS", "OS", "Com", "Var", "Grd", "PV", "ChC", "Cnd", "CSJ") the functional icon selected using the "Item" selector is added to the diagram by pointing to an appropriate location and pressing the left mouse button once. While the mouse is being moved to the correct location a wire frame tracking rectangle of a size appropriate to the icon being placed follows the mouse cursor. Multiple instances of the same functional icon may be placed with consecutive button pushes. If the "Prc" mode is selected, the Process Icon currently selected in the process browser is placed. Other options place different functional icons.

For links ("Item" modes "Ctr" and "Ch") the left mouse button is pressed once at the starting port and once at the finishing port of the link. Intermediate "joints" in the connection may be defined using further mouse button presses, with the link following the points in their order of selection. While links are being defined, the position of the mouse is tracked with an appropriate line for the link being defined. Connections may be cancelled by pressing the right mouse button.

6.3.1.3.1.2 Deleting diagram components

Diagram components may be deleted with the "Add or Delete Item" mode control in "Delete" mode and the "Edit :" mode control in "Diagram" mode. Confirmation of deletion requests is asked for by the system using a small pop-up window.

For Icons ("Item" modes "Prc", "IS", "OS", "Com", "Var", "Grd", "PV", "ChC", "Cnd", "CSJ") the diagram component of the type indicated using the "Item" mode control is removed from the diagram by pressing the left mouse button over it. Any connected Channel Links and Control Flow Links are removed when a diagram component is deleted. Multiple components may be deleted using consecutive mouse button pushes.

In link deletion, the left button is depressed once over the starting port of the link and once over the finishing port of the link.

6.3.1.3.1.3 Editing the textual or diagrammatic definition of a Process Icon

The internal detail (the definition) of a Process Icon instance may be displayed for editing with the "Add or Delete Item :" mode control in "Add" mode. Process Icons whose internal contents are to be edited are selected using the left mouse button. If the selection has taken place with the "Edit :" mode control in "Text", text may be entered for the process concerned into a text editing window. If the selection

has taken place with the "Edit : " mode control in "Diagram" mode the Process Icon becomes the icon currently being edited, just as if it had been selected using the "Edit process graph" mode of the Process Icon browser. The operation is equivalent to going down a level in the hierarchy of diagrams and provides a convenient way of graphically "zooming in" on areas of interest. For "Diagram" mode selections on textual Process Icons a pop-up window appears allowing the user to change the type of the Process Icon from textual to graphical. Similarly, for "Text" mode selections on graphical Process Icons a pop-up window appears which allows the user to change the type of the Process Icon to text. As all Process Icons are initially created as graphical Process Icons, this pop-up window provides a mechanism for the creation of textual Process Icons.

6.3.1.3.1.4 Editing the contents of text areas

The contents of text areas may be modified by placing the mouse over the area in any mode and clicking the left mouse button. Standard keyboard commands may be used.

6.3.1.3.1.2 Toggle controls

Two of the three "Toggle" items are used to modify the information displayed in the diagram editing area - the "Control flow on this diagram is" item and the "Stub Grid" item. The third item determines the procedural or non-procedural nature of the definition diagram being edited.

Control flow on this diagram is (cyclical choice item)

When the item is enabled ("On", the default condition) all Control Flow Links, Control Ports and Control Stubs in the definition diagram displayed in the diagram editing window are displayed. When the item is disabled ("Off"), no control flow information is displayed. The item supports the hiding of control flow information at particular levels of abstraction by remembering the value of the toggle for each diagram stored in the system.

Stub Grid (cyclical choice item)

When this item is enabled a grid showing the positions used in mapping Channel Stubs to Channel Ports is shown. The grid consists of three vertical and two horizontal lines, and is helpful for ensuring the placement of stubs in the correct locations.

Compile diagram as (cyclical choice item)

The "Compile diagram as" item determines the compilation algorithm to be used for the definition diagram currently displayed in the diagram editing area. If the choice is "as a procedure" the diagram is compiled as a procedure (for procedural Process Icons). If the choice is "as a code insert" the diagram is compiled as a series

of macro code inserts (for non-procedural Process Icons). This subject is dealt with in more detail in chapters five and seven.

6.3.1.3.1.3 System controls

The five systems function buttons in the diagram editor control panel are used to control the interaction of the diagram editor with the Sun's Unix file system, the GILT compiler and for exiting the system.

Compiler Window (cyclical choice item)

The "Compiler Window" button is used to display the compiler window. When the button is pressed, the window appears if it is not already displayed. The compiler is dealt with in the next chapter.

Diagram-Prolog (button)

The "Diagram-Prolog" button is associated with the compiler functions button and produces a file containing a set of Prolog clauses to be used with GILT's compiler if the compiler is not being used through the compiler control panel.

Load Diagrams and Save Diagrams (buttons)

The "Load Diagrams" and "Save Diagrams" buttons read and write files containing GILT diagrams. The filename to write to or read from may be highlighted using the Sun's selection facility (Sun, 1989).

Quit (button)

The quit button terminates the GILT editing system. Confirmation of the action is sought via pop-up windows before the action is taken, thus avoiding loss of user entered information.

6.3.2 The text editing system

GILT's text editing system provides facilities for the creation of textual specifications of Process Icon functionality. Text is entered into pop-up windows, which have surrounding areas for the definition of non-channel parameters (variables) and Channel Stubs.

6.3.2.1 Functional description of the text editor

Text windows are used to textually define the functionality of a Process Icon. A text window may be produced by clicking on an instance of a Process Icon in a diagram with the "Edit :" mode item in the diagram control panel set to "Text". Text

windows for up to four Process Icons may be simultaneously displayed. This limit is imposed by the number of available file descriptors per process under Sunview.

Text windows have already been described briefly in chapter five, which gave a high level overview of their functionality and defined the relationship between the components making up a text window and Occam processes. This section gives a lower level description of the functionality of text windows and explains the functional aspects of the windows in greater detail.

A text window is shown in Figure 6.8. Text windows have five functional areas. Central to each window is a text editing area. To the top and bottom of a text window are areas concerned with the definition of Channel Input Stubs and Channel Output Stubs. To the right of the central text editing area is the passed parameter declaration area. To the left is a control panel containing two buttons for finishing editing operations. The bar at the top of each text window displays a message showing the name of the Process Icon definition which is being edited using the window and, if a file is being edited, a filename. If a file is not being edited (as is the case with newly created windows) the message "Editing memory" appears in place of the filename.

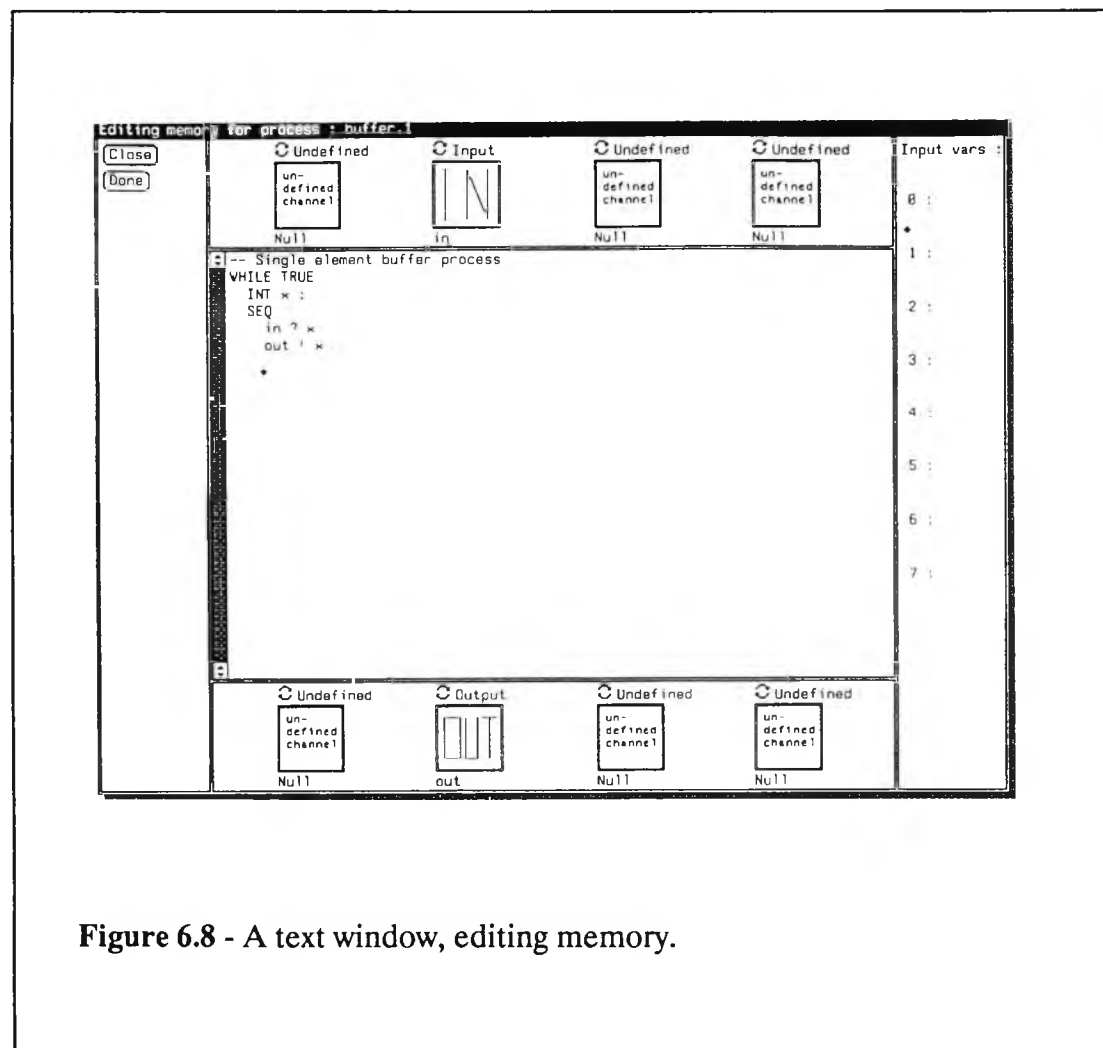


Figure 6.8 - A text window, editing memory.

The central text editing area has the same user interface as Sun's standard text editing system, `textedit`. In the window, a text cursor may be placed at any location by clicking with the left mouse button. Standard text editing keys may also be used to create a textual process description, as defined in chapter five. Using menus associated with the bar at the top of the window (accessed by holding the right mouse button down over the bar) editing facilities such as search and replace can be used, files external to the text editing system read in, etc. Full details of the user interface of the text editing parts of the windows may be found in (Sun, 1989).

The non-channel (variable) parameter declaration area to the right of the text editing window may be used in a similar fashion. Text strings as defined in chapter five may be typed into any one of the eight available positions by clicking the left mouse button over the appropriate position and typing from the keyboard. The same spatial parameter matching technique as is used for graphical Process Icon definitions is applied to Process Icons defined textually.

The control area to the left of the central text editing area contains two buttons, both of which are concerned with the closing the text window :

Close (button)

The "Close" button closes the text window, but does not save its contents. The window remains active and is displayed very quickly if activated by the selection of its representative Process Icon. Up to four windows may be open (displayed) or closed (not displayed, but still active) at any time. This limit is imposed due to the limitations on file descriptors available per process (64) under Sunview.

Done (button)

The "Done" button closes the text window, saves its contents and frees file descriptors and storage associated with it. The window does not remain active but may be recreated by appropriate selection of its Process Icon. The action increases the number of available windows by one, thus allowing text for more than four processes to be edited per session. The operation started by "Done" takes longer to perform than that initiated by "Close", especially if the editing system is being used on a slow machine (a 3/50 or 386i) as opposed to a faster one (a Sparcstation or one of the Sun-4 series).

The two stub definition areas (top and bottom) each contain four sets of three items. The top item in each item set is a cycle which is used to switch the type of the Channel Stub represented by the set from "undefined" (the initial state) to "input" or "output". When the cycle is selected by clicking on it with the left mouse button the central "stub icon" changes appropriately to show "IN" or "OUT".

The bottom item of each set, the "stub name" item, allows the entry of a textual label for the Channel Stub, which is used to refer to the stub in the text.

The central "stub icon" serves a dual purpose. It visually indicates the state of the Channel Stub (input, output or undefined) and allows the automatic insertion of Occam input or output processes into the text. When the stub icon is selected with the mouse by clicking the left mouse button, a reference to the stub is inserted into the text. The reference includes the stub's name (defined using the stub name item) and an appropriate Occam input or output directive ("?" or "!"). For example, pressing the stub icon named "in" in figure 6.8 would result in the insertion of the string "in ?" at the current cursor position in the text. The system frees the programmer from remembering the names of stubs and allows textual programming to proceed in a "spatial" manner.

6.4.2 Implementation of the GILT program editor

Reasons for the selection of Sunview as an implementation environment have already been discussed. This section analyses relevant features of Sunview and shows how it was used in implementation of the editing system. Details of the implementation are also discussed briefly. Only those features of the implementation which are considered to be of particular importance are discussed in detail.

6.3.2.1 Relevant features of the Sunview system

Sunview (Sun Visual/Integrated Environment for Workstations) is a user interface toolkit which supports interactive graphics applications on Sun workstations. Sunview allows software systems to be built out of a number of basic building blocks including four types of windows known as "canvases", "text sub-windows", "panels" and "tty sub-windows". Canvases provide high level raster based drawing areas, while text sub-windows have built in text editing capabilities. Panels contain user interface "choice" items such as buttons and sliders. Tty sub-windows allow programs to be run within them. A further class of windows, known as "frames", combine sub-windows of any of the types above to form larger windows. Frames are used to create applications consisting of many different windows.

Sunview's runtime system has a centralised window manager, which manages overlapping windows and distributes user input to the appropriate window applications. Events are internally distributed to application components by a "notifier" system. Components (such as windows, buttons and sliders) have system or user defined procedures handling their interactive functionality.

Sunview applications are coded in a callback style. The main control loop in a Sunview application resides within the notifier, not within the application. The notifier reads events and notifies, or calls out to, procedures which form the application and which have been registered with it ("callback procedures"). Registration takes place at the time of an item's creation. Figure 6.9 illustrates the typical flow of control in a callback based Sunview application, which consists of a number of callback routines which are executed by the notifier depending on the user input it receives. Callback based programming removes the burden of

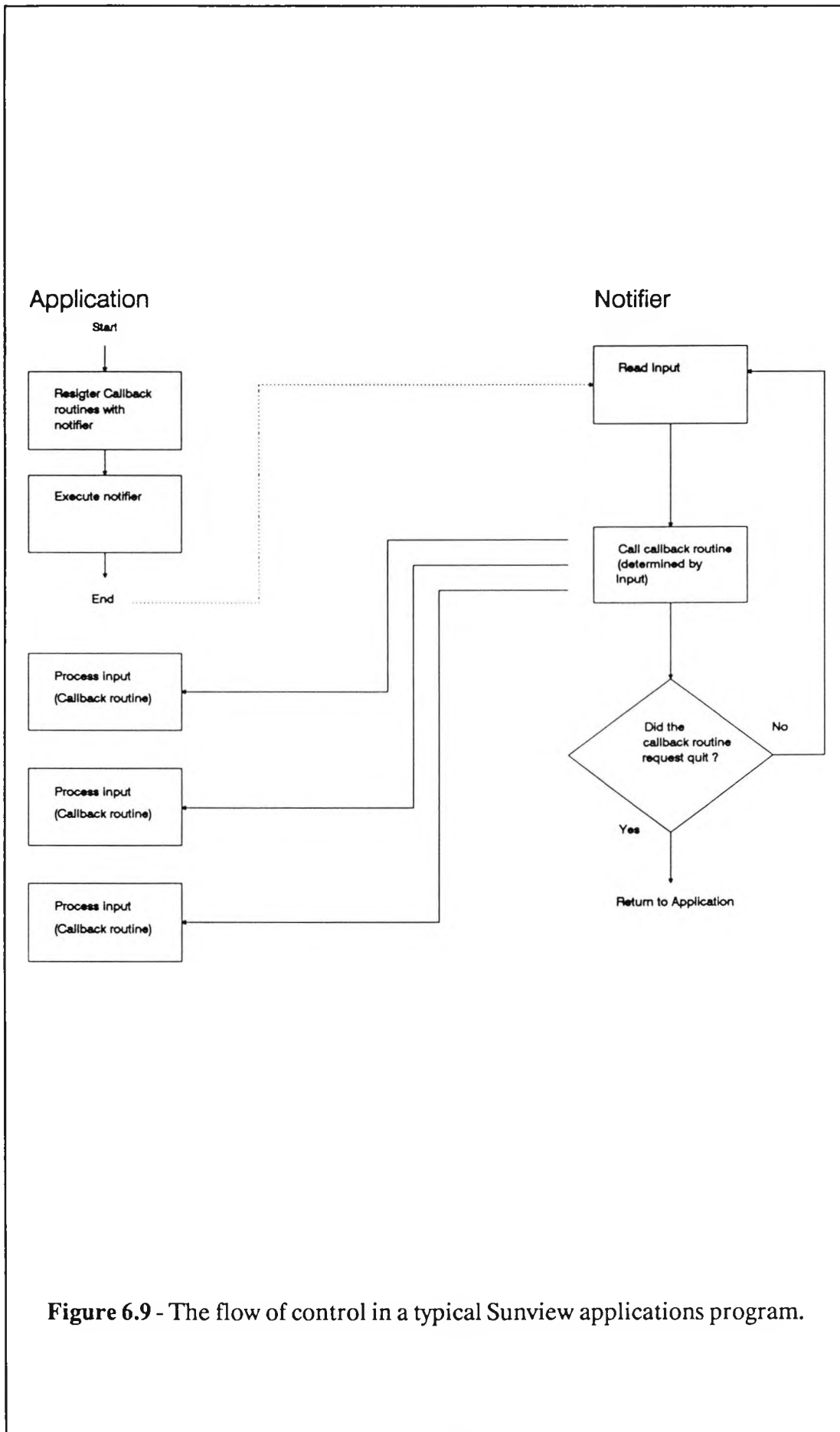


Figure 6.9 - The flow of control in a typical Sunview applications program.

managing a complex, event driven, environment from the applications programmer. This results in simpler and smaller applications programs.

Library routines are provided for the creation, destruction and modification of windows, buttons and other user interface components. Extensive use of these routines is made in the GILT diagram editor, which uses buttons to form parts of functional icons. Sunview windows may also be accessed at a lower level using basic raster manipulation library routines, which are used by the GILT editor for functions like drawing links between functional icons.

As mentioned above, user interface items or "widgets" are created using library routines. At creation, properties of items (for example their x-y position within a window and their size) are set using a number of attributes in a variable length attribute list. Each attribute consists of a mnemonic constant (for example "PANEL_ITEM_X", followed by a number (possibly zero) of parameters. Most attributes for an item are readable and writeable at any time, with library routines provided for this purpose. Two possible list attributes are used to register an item with the notifier. These attributes ("PANEL_NOTIFY_PROC" and "PANEL_EVENT_PROC") may only be set at creation time. Another attribute, "PANEL_CLIENT_DATA" may be used to hold application specific data.

Example

The call :

```
button = panel_create_item(window, PANEL_BUTTON,
    PANEL_ITEM_X, 100,
    PANEL_ITEM_Y, 200,
    PANEL_BUTTON_IMAGE, button_image,
    PANEL_NOTIFY_PROC, button_proc,
    PANEL_CLIENT_DATA, 99,
    0) ;
```

Creates a button "button" in the window "window" at co-ordinates (100,200). The item's image is pointed to by "button_image". When the button is selected (using the left mouse button) the routine "button_proc" is called. The item attribute "PANEL_CLIENT_DATA" is set to have the value "99". The example uses the "PANEL_NOTIFY_PROC" attribute for event handling. The "PANEL_EVENT_PROC" attribute provides a lower level interface to the event handling procedure than does the "PANEL_NOTIFY_PROC" attribute, but its usage is almost identical.

Similar routines are provided by the Sunview libraries for the creation of windows of different types.

The attributes of items and windows may be read using library routines.

Example

```
pcd = panel_get(item, PANEL_CLIENT_DATA);
```

may be used to read the item's "PANEL_CLIENT_DATA" attribute.

Callback procedures, which are associated with items, have a defined type and number of specified parameters. As an example a callback procedure for the earlier button is shown below :

Example

```
static int
button_proc(item, event, value)
Event *event;
Panel_item item;
Int value;
{
    int pcd;
    pcd = panel_get(item, PANEL_CLIENT_DATA);
    print("The item that you have just selected has
attribute\n");
    printf("PANEL_CLIENT_DATA set as  %i.\n", pcd);
}
```

In the procedure, which is called whenever the button is pressed, the "PANEL_CLIENT_DATA" attribute is read from the data structure describing the item and printed out.

Similar event procedures for windows may also be created. Data describing the nature of the event which lead to the calling of the procedure, such as the x-y co-ordinates of the mouse, and the event type, may be read in a similar fashion to the methods used for reading the "PANEL_CLIENT_DATA" attribute.

6.3.2.2 The Process Icon editor (implementation)

The Process Icon editor uses a number of callback routines associated with the items in the control panel to set variables indicating the mode of the mouse within the drawing area. Routines associated with other items, such as the "Clear" button perform specific image manipulations of the drawing area and associated icon image.

A callback routine for the drawing area processes user interface events, such as mouse button presses, and interprets them according to the mode variables set by the control panel items. Feedback is provided using Sunview's drawing routines and all instances of the Process Icon being modified are updated appropriately.

A further callback routine, called when the mouse is in the Name Text Areas of a Process Icon, updates Process Icon names.

A Process Icon's image is stored in an area of memory which is dynamically allocated when the icon is created in the Process Icon library or when files containing descriptions of GILT programs are read into the system.

6.4.3.3 Implementation of the Process Icon library (browser)

The Process Icon library is implemented in a similar manner to the icon editor. Some items in the control panel have callback routines which perform specific functions like creating icons, or hiding the Process Icon browser window. Other items are used to determine the action to be taken when a Process Icon in the main part of the window is selected. Process Icons in the main part of the Process Icon browser are implemented as pairs of items (a text item and a button which has as its image the image of the Process Icon which it represents) with associated callback routines.

User interface items representing Process Icons are dynamically created and destroyed within the Process Icon library as users of the system add and delete processes. Each Process Icon definition contained in the library is represented by a button showing the Process Icon's image and an associated text area. The `PANEL_CLIENT_DATA` attribute of each Process Icon item is used by the item's callback routine to reference an array which contains a pointer for each Process Icon in the browser (the Process Icon look up table). Pointers in the array reference data structures known as "definition nodes". Figure 6.10 shows a diagrammatic representation of this system.

Definition nodes define the structure of Process Icons and provide references to the component parts of their definition diagram or to a text file containing their textual specification. Each definition node contains information like the number and relative x-y locations of Channel Ports and pointers to items defining the Process Icon's image and name. The internal detail of a graphical Process Icon is described using data structures called "instance nodes", which are fully discussed in section 6.4.3.4. Arrays of pointers within definition nodes reference instance nodes which represent functional icons and the links between them. Two sets of these pointers are used in a definition node. The first (instance-definition pointers) contains the addresses of all instances of the Process Icon defined by a particular definition node, and is used to ensure that changes made to the Process Icon through editing are correctly propagated to all of its instances. The second (parent-child pointers), which only exist for graphical Process Icon definitions, reference all of the instance nodes which define the Process Icon's functionality. Textual Process Icons make no use of the last set of pointers, instead having a textual filename and a set of numeric values storing their Channel Stubs and non-channel (variable) parameters. Figure 6.11 shows the relationship between instance nodes and definition nodes via the pointers mentioned above.

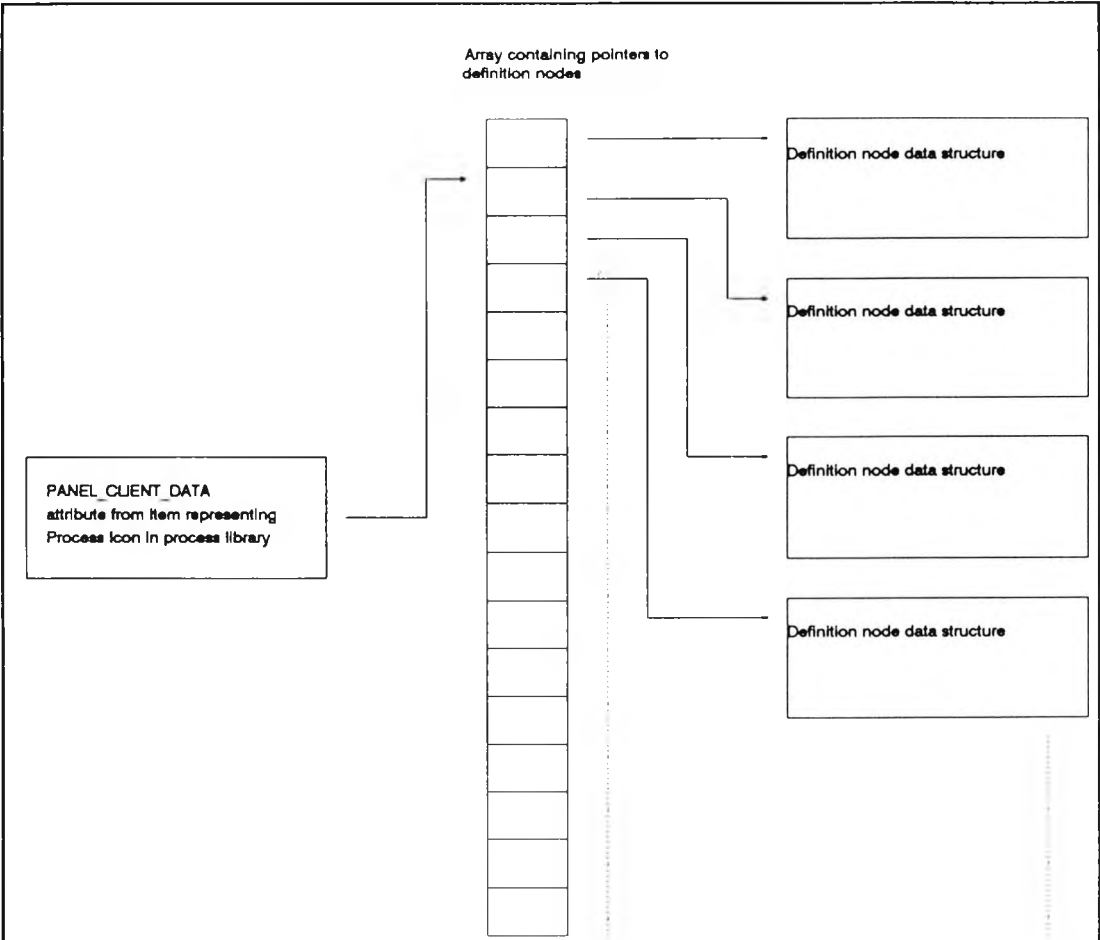


Figure 6.10 - How items in the browser reference Process Icon definitions stored in definition node data structures. Lines indicate pointer references.

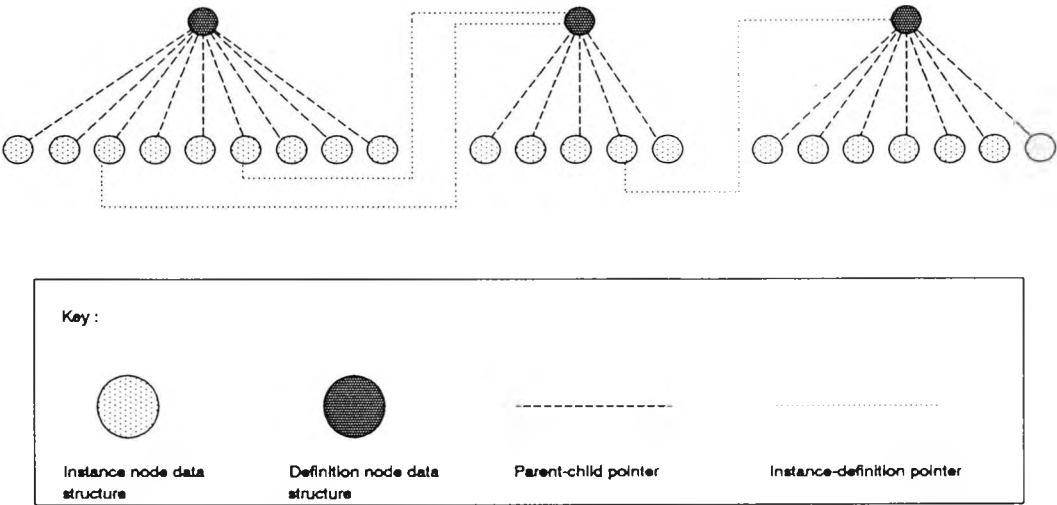


Figure 6.11 - The pointers between definition nodes and instance nodes.

When a Process Icon is created using the "Add Process" button a definition node is created, initialised, and linked to a Process Icon look up table. Instance nodes describing a Control Flow Input Stub and a Control Flow Output Stub are also created and linked to the definition node to define the control flow entry and exit points of the diagram. No control flow links are made to or from the two icons, which define a newly created icon to be a stop process. Finally, items representing the Process Icon are created in the main window of the Process Icon library.

6.3.3.4 Implementation details of the diagram editor

GILT diagrams are represented in the diagram editor by a dynamic self referential C data structure, created from instance nodes and definition nodes which have already been briefly discussed. Sunview items (such as buttons and text editing areas) are used to represent functional icons. Links between components are drawn using routines which access windows at a lower level than the interface provided by the Sunview items.

The instance node data structure is used to describe functional icons and contain information like the x-y position and type of a functional icon. Data describing Control Flow and Channel Links between the functional icons is also included. The logical or connective aspects of Control Flow and Channel Links are described by sets of pointers which reference other instance nodes. This information knits together instance nodes into constructs. Further pointers refer to simple data structures storing the physical co-ordinates for the links. Pointers are also included to link the instance node with a definition node which stores information on the Process Icon definition of which the instance node is a part. Instance nodes which represent Process Icons also have an additional pointer which references the definition node defining the Process Icon, as discussed in section 6.4.3.3.

As mentioned above, links between the ports of functional icons are implemented as pointers between instance nodes. Each instance node contains a number of arrays of pointers each of which contains pointers for outgoing and incoming links of different types. Two functional icons with a connecting link each have a pointer to the other and additionally have a pointer to a separate structure which contains the physical co-ordinates of the link. Figure 6.12 shows how a Channel Link is represented in the data structures. The representation used for Control Flow Links is similar, but an added complication is introduced by the need to distinguish between links connected to Control Flow Output Ports and Not Control Flow Output Ports. This is dealt with by a subscripted array which indicates which of the control flow pointers deal with links from Not Control Flow Output Ports.

Sunview's items provide a very easy implementation path for creating diagram editors like the one used by GILT, and cut down on the time required for implementation greatly. Although the items were originally designed for providing control functions in Sunview applications, the implementation of GILT has shown that they may be equally well applied to the representation of components in diagrams and the provision of a flexible diagram editing system.

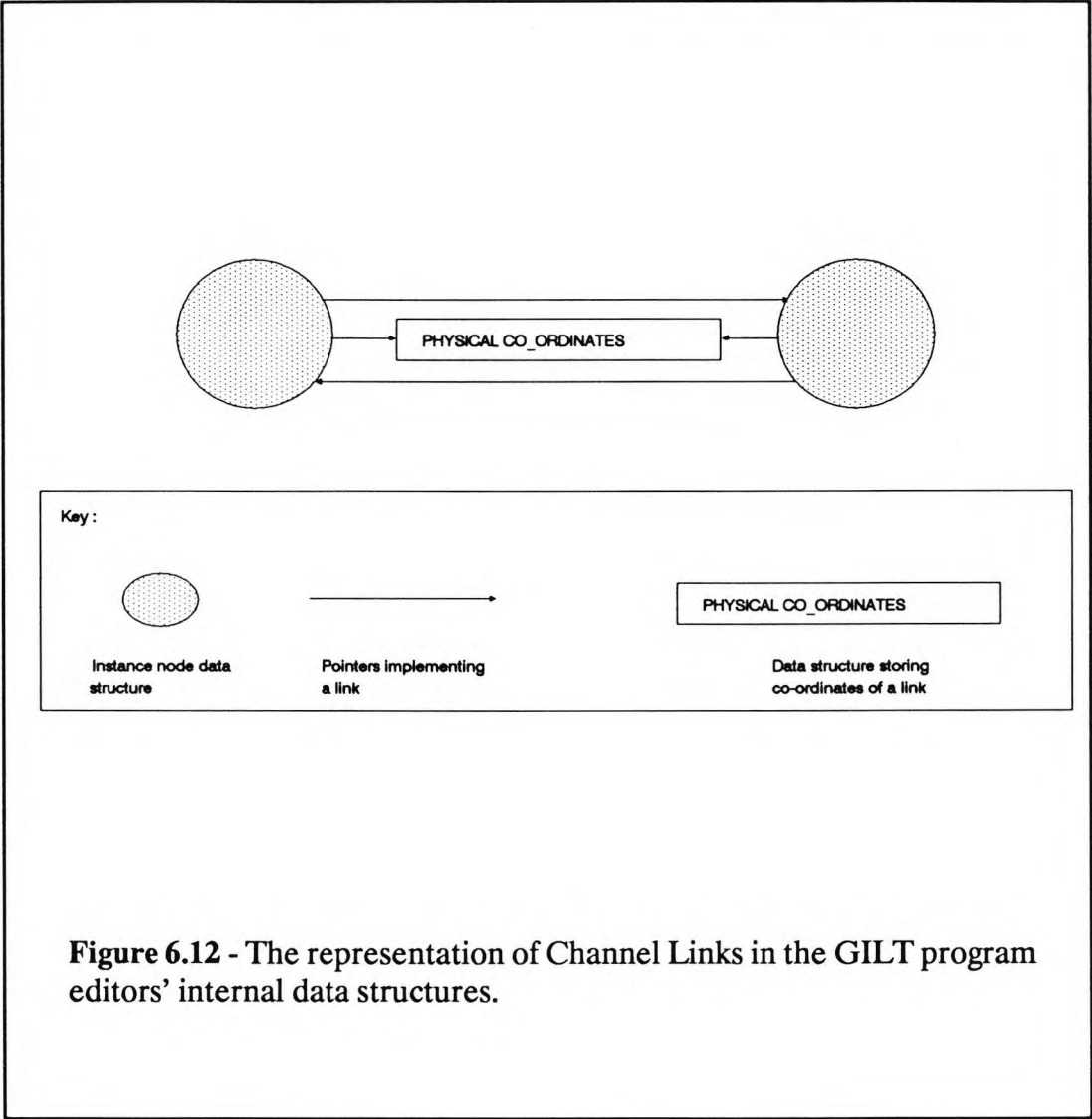


Figure 6.12 - The representation of Channel Links in the GILT program editors' internal data structures.

Each type of component making up a functional icon is represented by a different type of item with an associated callback routine. All components of a particular type are registered with the same callback routine. Like the items representing Process Icons in the Process Icon library, the "PANEL_CLIENT_DATA" attribute of every item in a diagram is used to refer to a data structure. The structure, instances of which are dynamically created when new items are created, holds information on the type of component that an item represents and a pointer to an instance node describing the functional icon of which the item is a part.

Figure 6.13 shows how many separate item descriptors reference the same instance node data structure through multiple "button_descriptor" data structures.

The mode controls in the control panel set global mode variables for the program panel. The global mode variables determine how events are processed by a callback routine associated with the diagram editing area. For example, if the mode control items are set for the addition of a Process Icon, then a new instance of a Process Icon is created at the x-y co-ordinates of a left mouse button push. When

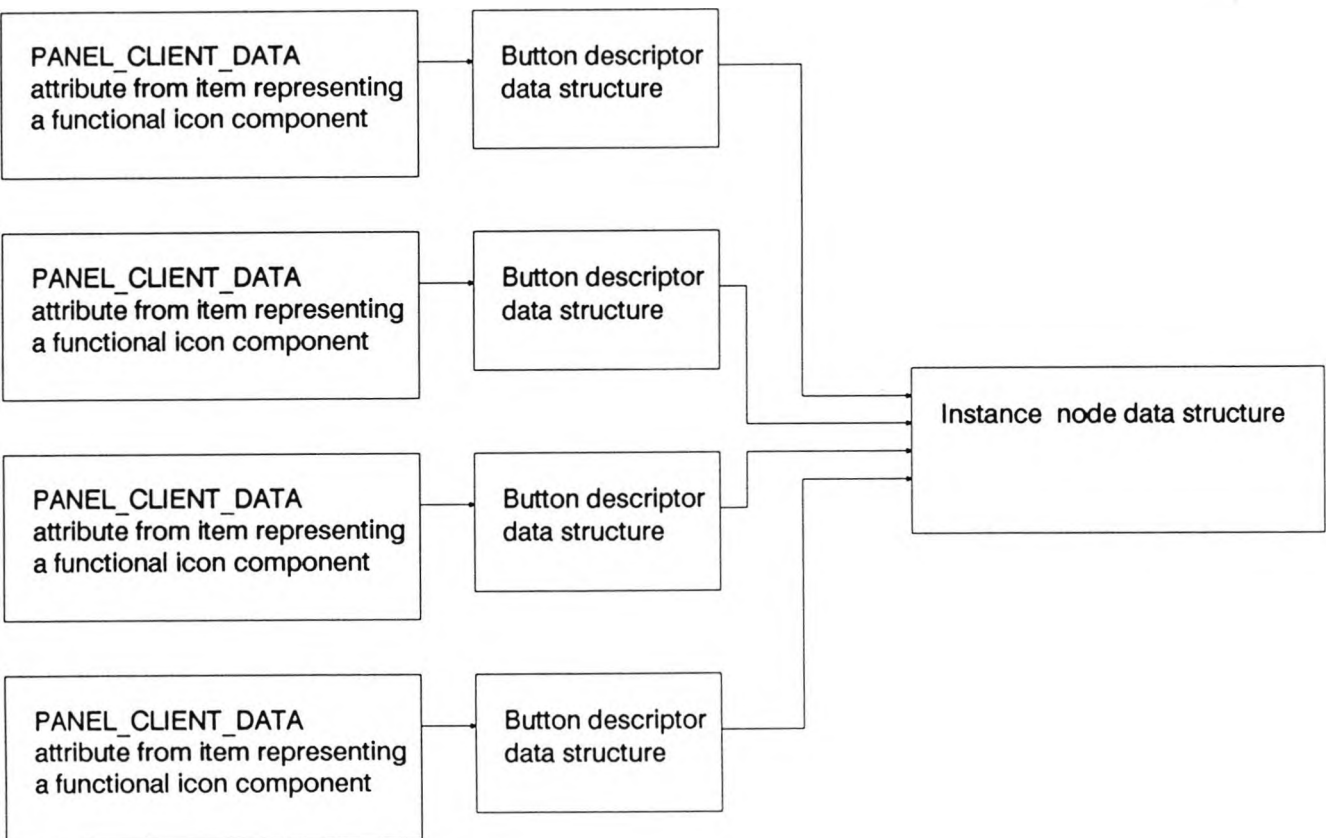


Figure 6.13 - Separate item descriptions referencing the same instance node data structure through multiple button descriptor data structures. Thin lines indicate pointer references.

such a functional icon is "placed" a new instance node describing the icon is created and linked appropriately into the dynamic data structure. Representative items for the component are created with `PANEL_CLIENT_DATA` attributes referring to the newly created instance node.

The insertion of links between components corresponds to pointer assignment operations and the creation of a new structure to hold the physical co-ordinates of the link.

Deletion operations are implemented by memory deallocation routines with associated pointer reorganisation and screen redrawing.

The networks of connected instance and definition nodes are saved and loaded with the aid of special routines which "flatten" and "unflatten" the data structures to and from linear files suitable for storage in the Sun's Unix file system. All the routines are accessed through callback routines associated with the system function buttons in the diagram editor control panel.

The diagram editor performs lightweight checking while diagrams are being input. By the use of simple restrictions encoded in the callback routines very many illegal structures are precluded. Major restriction are obtained by ensuring that functional icons are input as complete diagrammatic units, not as their separate component parts, which would require much more checking to be performed in the parsing sections of the compiler than is currently performed. Further simple restrictions in the routines concerned with the entry of links prevent the connection of Channel Links to Control Flow Ports or Stubs and vice-versa.

As the grammar for the communications structures is extremely simple sufficient restrictions, encoded as conditionals, are placed on the entry of channel connections to ensure that only Channel Links which might form part of legal communications structures may be entered. The editing system allows the entry of Channel Links with one end (and one end only) connected to a Channel Connector Icon, but rejects all other, clearly illegal, connections. Only one link may be connected to a Channel Input Port or Output Stub, or from a Channel Output Port or Input Stub. Multiple links may however be connected into or out from a Channel Connector Icon. The restrictions imposed are context free in that the decision taken by the system to allow the entry of a Channel Link between two functional icon components is dependent only on the type of the components.

It is more difficult to devise simple restrictions which preclude the entry of incorrect links in more complex diagrammatic structures such as GILT's control flow structures. Any such restrictions would need to take account of the context of a component. One possible approach would be the provision of a interactive parser, which would check diagrams to see if connections or components could form part of a legal structure and accept or reject them accordingly. Such a system would require the use of algorithms similar to those used in the compilation system, but would need to be able to distinguish between clearly incorrect structures and those which are potentially correct. It would not be able to preclude

the input of all incorrect diagrams, as any diagram which does not contain complete constructs (for example those in the process of being constructed) is incorrect.

6.3.3.5 Implementation of the text editor

The implementation of the text editor was far simpler than that of the graphics editor. Sunview's standard text editing library routines are used for the central areas of the text editing windows, while callback routines associated with user interface items provide the functionality for items representing Channel Stubs and non-channel parameters. The callback routines update a definition node data structure, associated with the window by pointers, which holds information on a textual Process Icon definition. Further callback routines handle functionality of the "Done" and "Close" buttons. The contents of the central text editing area is stored in a file, which is referenced by the definition node data structure for the textual Process Icon definition which is being edited in a text window. Each textual Process Icon has a separate file within the Unix file system, with each file having a unique name generated by a special routine.

A Compiler for the GILT language

7.0 Introduction

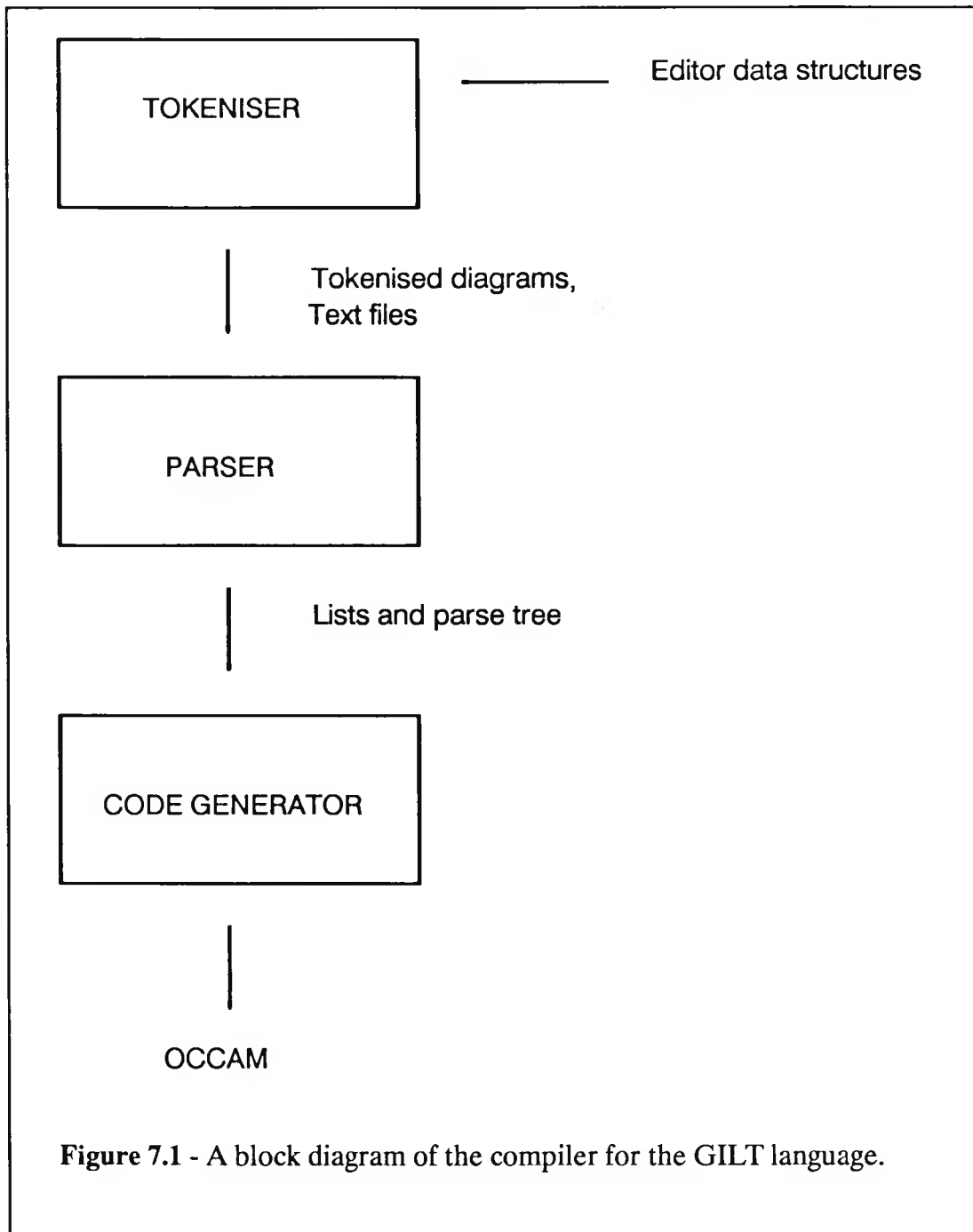
This chapter describes the compilation system built for generating Occam code from GILT programs. Occam was chosen as a target language because it formed the basis of GILT's constructs, allowed the code generation sections of the compiler to be far simpler than would ordinarily be the case, facilitated easy human analysis of the output code from the compiler, and was already used for the functional specification of textual Process Icons. Alternative approaches, such as generating native Transputer instruction code, would not allow such easy analysis of the output code or the execution of GILT programs on non-Transputer architectures supporting Occam compilers.

7.1 Previous work on compilation systems for visual languages

Little work has been published on compilation systems for visual languages as most previous visual languages have been interpreted, and not compiled. Lakin (1987) is the most often quoted work in the area and is concerned with the understanding of generalised diagrams through "spatial parsing" using a form of context free grammar. In (Gillet and Kimura 1986a) and (Gillet and Kimura 1986b), the parsing of the visual language "Show and Tell" is discussed. A compiler for the visual language "Prograph" is discussed in (Cox and Mulligan 1985). Work on both of these languages does not use a formal visual grammar like GILT's but instead uses an approach based on a textual representation of the diagram being compiled. The set of possible textual representations of the program is defined using a textual formalism, leading to a compiler significantly different from the one produced for GILT. No compiler for a visual language resembling GILT has previously been produced.

7.2 Compiler overview

The compiler has an overall structure similar to that of a compiler for a conventional textual language, having a tokeniser, a parser, and a code generator. A block diagram of the system is shown in figure 7.1. The main difference between GILT's compiler and conventional compilers lies in the nature of the tokeniser and parser.



The tokeniser, which may be regarded as forming part of the editor, is far simpler than a tokeniser for a conventional language because the bulk of the tokenisation work is performed by GILT's diagram editing system. The tokeniser converts information stored in the editor's internal data structures into a more explicit form suitable for input to the parser. Tokens output from a tokeniser for a visual language must be significantly different from those generated for a textual language to take into account the multi-dimensional nature of visual languages. GILT's tokeniser outputs graphs, represented by Prolog facts, which have a direct relationship to the graphs of the syntax given in chapter five. Prolog facts provide a convenient medium for the transfer of information between the tokeniser and

the remainder of the compilation system, and have the expressive qualities necessary for representing the diagrams.

Because of the fundamental differences between graph and text grammars it is not possible to use established textual parsing algorithms for the implementation of the parser. A very general parsing algorithm for two-dimensional grammars similar to GILT's is therefore presented in this chapter. The algorithm, which is computationally similar to the limited backtrack bottom up parsing algorithm used for some textual languages, has been implemented in Prolog to form the basis of GILT's parser. An explicit parse tree is generated by the parser.

The code generator, which produces Occam from the parse tree generated by the parser, is written in Prolog but has a conventional structure.

The user interface to the compilation system is written in C and is integrated with the editing system to provide a comprehensive system for the display of error messages and control of the compiler's functions. A facility to relate error messages back to erroneous features in diagrams is also provided by the user interface.

7.3 Choice of implementation language

Prolog was chosen as the implementation language for most of the compiler for the following reasons :

- 1) It provided a natural medium for the expression of graph structures using Prolog facts, which allow the multi-dimensional nature of GILT diagrams to be expressed.
- 2) Prolog's extra logical functions (assert and retract) provided a natural implementation of the graph reduction operations required by the parser.
- 3) The backtracking mechanism provided a clear way of implementing recogniser routines in the parser.
- 4) Prolog's dynamic, recursive nature removed the need for complex data structures storing information, allowing instead the use of simple Prolog lists.
- 5) The use of Prolog allowed interactive testing of the individual Prolog functions making up the compiler.
- 6) A very good implementation of Prolog (Hutchins, 1986) was available on the machine used for development of the system.

Overall, Prolog supported a very clear expression of the ideas and algorithms used in the compiler, and so was an ideal implementation language for the prototype compiler.

7.4 A very brief introduction to Prolog

Prolog (Clocksin and Mellish, 1981) is a logic programming language. Prolog programming consists of defining and querying relations, with a Prolog program consisting of a number of "clauses". There are three types of clause; "facts", "rules" and "questions". A relation can be specified by facts, which simply state that n-tuples of objects satisfy the relation, or by stating rules about relations. The arguments of relations can be (amongst other things) concrete objects, called atoms, or variables. A Prolog procedure is a set of clauses concerning the same relation. Querying relations, by means of questions, resembles querying a database, with Prolog's answer to such questions consisting of a set of objects which satisfy the queries.

Prolog establishes whether an object satisfies a query by logical inference, by exploring alternatives and by backtracking. These processes are performed automatically by the Prolog system and are (at least in principle) hidden from the user.

A number of other facilities are provided by Prolog, such as arithmetic operations and functions for the maintenance of Prolog's internal database.

Two types of meaning of Prolog programs are generally distinguished; declarative and procedural. The declarative meaning is advantageous from the programming point of view, and is concerned only with the relations defined by the program. It determines what the result of a program will be. The procedural meaning determines how the result is obtained and how the relations are actually evaluated by the Prolog system.

The following example provides a quick overview of some of Prolog's logical facilities :

Example

The rule

```
fallible(X) :- man(X).
```

expresses the fact that "All men are fallible", by reference to the variable X.

```
man(socrates).
```

expresses the fact "Socrates is a man" by reference to the atom "socrates".

A question, or goal, may be put to the Prolog system to test its powers of logical inference :

```
fallible(socrates) .
```

Prolog evaluates the goal by logical inference. In this case the goal would succeed and, if typed interactively at a keyboard, would return "yes". Further Prolog facilities are introduced as required in the following sections.

7.5 The tokeniser

The previous chapter presented details of the internal storage of GILT diagrams. The diagram editor's internal data structures store the graphical parts of GILT programs. The tokeniser, which is written in C, converts the internal representation into a form suitable for input to the parser. It is far simpler than a corresponding tokeniser for a textual language and performs a simple translating action. The tokeniser produces tokens by iterating over all of the data structures which store a GILT program in the editor and outputting Prolog facts according to a set of simple conversion rules encoded as conditionals. Though the tokens are output in Prolog, they could easily be output in some other form. For example, were the tokeniser's output to be processed by a parser not written in Prolog, little would be gained from outputting tokens in Prolog. A better approach would be to output them in a more compact, textual or perhaps tabular format.

A unique identifier, expressed as a Prolog atom, is produced for every item of interest in a GILT program. Each identifier refers to a separate item. A variable number of tokens for each item are expressed using Prolog facts. The number of tokens produced for an item depends on the type of the item, with each token declaring a separate fact about an identifier. All the tokens for a particular item of interest refer to the same identifier which is generated from the address of the C data structure used to represent the item in the editor. For example, if the address of the C data structure storing information on a Process Icon instance was "480440", then the token below would be output :

```
process_instance_node(480440) .
```

The tokens referencing a single identifier form nodes in graphs whose edges are formed by tokens which express relationships between two identifiers. For example, a Control Flow Link between the Control Flow Ports of two functional icons assigned the identifiers "480440" and "500040" would be represented by the token :

```
control_connect(480440, 500040) .
```

Tokens expressing relationships between identifiers are also used to denote Channel Links between diagram components and to describe the hierarchical nature of GILT diagrams.

Two types of identifiers, "definition identifiers" and "diagram component identifiers", are produced by the tokeniser. Definition identifiers are used to reference information concerning Process Icon definitions, while diagram component identifiers are used to reference information about functional icons or functional icon components, such as Process Icon instances and the connections between them. Both of the examples given above are of diagram component identifiers. "Hierarchy tokens" are used to express the hierarchical nature of GILT diagrams by linking together definition identifiers and diagram component identifiers via simple relationships similar to the earlier example which showed how Control Flow Links are represented.

7.5.1 Definition identifiers and associated tokens

A definition identifier is generated for every Process Icon definition stored in the editor. A set of "definition tokens" are output by the tokeniser to describe a Process Icon definition by reference to the definition identifier. Collectively, all of the definition tokens form a definition node. Each definition token is a simple translation of information held by the editor in a definition node data structure (described in section 6.4.3.4). Definition tokens are not produced for every item of data held in the structure because much of the data is not relevant to the compilation process and may be discarded. A good example of such discarded information is the pointer which refers to a Process Icon's image.

Example

Assuming that the unique identifier generated for a Process Icon definition (the definition identifier) is the atom "450656", the Process Icon definition would be described by the following sequence of definition tokens, which together form a definition node :

```
process_definition(450656).  
process_definition_name(450656, 'process.1').  
compile_as_procedure(450656).
```

Each of the three tokens, expressed as a Prolog clause, defines a fact about the identifier "450656". The first clause defines the identifier "450656" to represent a Process Icon definition. The second clause defines the name of the Process Icon definition represented by the identifier to be "process.1". The third states that the Process Icon definition should be compiled as a procedure, not as a macro. Were the Process Icon definition to be compiled as a macro, the token "compile_as_code_insert(450656)." would replace the third token.

For a textually defined process a number of additional tokens are also output. These tokens define attributes for a Process Icon definition such as the filename of the file containing its Occam definition, its declared non-channel (variable) parameters and its Channel Input and Output Stubs.

Example

```
process_definition(503752).  
process_definition_name(503752, 'buffer.1').  
compile_as_procedure(503752).  
text_filename(503752, '/home/cb538/gilt/GLTAAa00622').  
declared_input_channels_list(503752, ['in']).  
declared_output_channels_list(503752, ['out']).  
declared_variables_list(503752, []).
```

The above tokens define facts about the identifier "503752" and together make up a definition node. As before, the tokens define a name and a compilation method for a Process Icon definition. As the definition of the Process Icon is textual, four extra clauses are output. The first provides a filename which contains the text for the process, with the three subsequent tokens defining parameters for the Process Icon definition. Together, all the tokens give a specification for a buffering Process Icon named "buffer.1" whose textual specification is contained in the file "/home/cb538/gilt/GLTAAa00622". The Process Icon has a single Channel Input Stub named "in", a single Channel Output Stub "out", and no passed non-channel parameters. The order of the parameters in the lists reflects their spatial positioning.

7.5.2 Diagram component identifiers and associated tokens

For graphical Process Icon definitions, tokens are produced for each functional icon and link forming part of the Process Icon's definition diagram. Dependent on the type of the functional icon a variable number (usually one) of "diagram component identifiers" are produced, each identifying a separate component of the icon. A number of "diagram component tokens" for each identifier are output, with each token describing a different facet of the component represented by the identifier. Taken together all of the diagram component tokens referring to a particular identifier will be termed a "diagram component node". All the diagram component tokens making up a diagram component node are translations of information held by the editor's instance node data structures (section 6.4.3.4). As in the earlier translations of information held in definition node data structures, some stored information is irrelevant to the compilation process and is not tokenised.

Each diagram component node represents an element of GILT's syntax, such as a functional icon. The diagram component nodes form the nodes of graphs, with edges being formed by relationships between them. Edges between diagram component nodes are equivalent to links between nodes in GILT's syntax.

Most functional icons are described using a single diagram component node consisting of an diagram component identifier and a few related tokens. This reduces the amount of information transferred between the tokeniser and the parser as opposed to describing every functional icon by a set of tokens representing the network of terminal symbols describing it in the grammar. The

result is a smaller, more efficient, parser which has to perform fewer parses over its input in order to generate a parse tree than would be the case if most functional icons were described with multiple diagram component nodes. Multiple diagram component nodes are used to describe functional icons which have Channel Ports. Channel Ports need to be tokenised separately from the bulk of a functional icon's components so that the communications structure of a GILT diagram may be parsed separately from the rest of the GILT diagram.

Multiple diagram component nodes describing a single functional icon are combined into a single coherent structure using "link tokens". Each link token refers to two diagram component identifiers and is used to link a diagram component node representing, for example, a Channel Port (a "peripheral node", with a "peripheral identifier") to a diagram component node which represents the body of the functional icon (a "central node", with a "central identifier"). Link tokens are also used to express the Channel and Control Flow Links between functional icons as edges between diagram component nodes. No distinction is made between link tokens representing part of the internal structure of functional icons and link tokens representing links between functional icons.

There is a direct equivalence between the tokens output for a functional icon and the functional icon's representation in GILT's syntax. Every functional icon is represented by a unique set of instance tokens. Table 7.1 (next page) gives an equivalence between symbols in GILT's grammar and the tokens output. Tokens completely describe the diagram for which they are produced but do not model the diagrams' syntax to the level of detail in the syntax of chapter five. The output from the tokeniser may be thought of as describing diagrams for a smaller syntax than the syntax of chapter five, but one which is equivalent and equally expressive.

Example

Guard Icons are described using tokens referring to two diagram component identifiers - one central identifier and one peripheral identifier. Assuming that the central and peripheral identifiers produced by the tokeniser are "466312" and "466322", the following sequence of instance tokens and link tokens describing the Guard Icon would be produced :

```
guard_node(466312) .  
guard_text(466312, 'down') .  
guard_text_2(466312, 'accept.dn') .  
channel_input_port(466322) .  
channel_connect(466322, 466312) .
```

The tokens collectively describe a Guard Icon, as defined by the non-terminal symbol "GUARD" in the grammar of chapter five, and provide details of the contents of the Guard Icon's two text areas. The tokens referring to the Central Identifier "466312" model the central part of the Guard Icon (non-terminal symbol "GUARD BODY" in the syntax) while those referring to the peripheral identifier "466322" model the guard's Channel Input Port (terminal symbol "CHANNEL

Syntactic element	Prolog Facts Output
CONTROL IN STUB(T)	control_in_stub(i).
CONTROL OUT STUB(T)	control_out_stub(i).
CHANNEL INPUT STUB (T)	channel_input_stub(i). parameter_order(i,n).
CHANNEL OUTPUT STUB(T)	channel_output_stub(i). parameter_order(i,n).
COMMENT (T)	comment(i) comment_string(i, 's').
VARIABLE DECLARATION (N)	variable_declaration(i). variable_declaration_text(i, 's').
GUARD BODY (N)	guard_node(i) guard_text(i, 's'). guard_text_2(i, 's').
CHANNEL INPUT PORT (T)	channel_input_port(i). channel_connect(i,i).
CHANNEL OUTPUT PORT (T)	channel_output_port(i). channel_connect(i,i).
PASSED PARAMETER (N)	pvar_node(i). parameter_order(i, n). variable_text(i, 's').
DECLARED PARAMETER (N)	dvar_node(i). parameter_order(i, n). variable_text(i, 's').

Table 7.1 - The equivalence between the valued nodes of GILT's syntax and the Prolog facts or tokens which are output to represent them. The table is continued on the next page, with explanation.

CHANNEL CONNECTOR (T)	chancon_instance(i). channel_name(i, 's').
PROC MIDDLE (N)	process_instance_node(i). instance_of_definition(i,d).
COND (N)	condition_node(i). condition_text(i, 's').
CONTROL SPLIT JOIN (N)	control_split_join_node(i).

Table 7.1 - The equivalence between the valued nodes of GILT's syntax and the Prolog facts or tokens which are output to represent them. Only tokens representing diagram components are included in the table. The terminal or non-terminal nature of the valued node in the syntax is indicated by "(N)", for non-terminal or "(T)", for terminal. Each token has a number of associated arguments, enclosed in brackets. The following letters are used to indicate the nature of the arguments :

i- An instance identifier, a unique tag assigned to relate all of the facts referring to a particular syntactic element together. For the "channel_connect" tokens output for terminals "CHANNEL INPUT PORT" and "CHANNEL OUTPUT PORT" two instance identifiers are indicated, showing that diagram component nodes for Channel Output and Input Ports must always be connected to (or from) a central diagram component identifier.

n- A general integer number, mostly used to indicate the number used in "parameter_order" tokens.

s- A general string. The strings are derived from the contents of the text areas forming part of functional icons.

INPUT PORT"). The link token "channel_connect(466322, 466312)" connects the two sets of tokens via reference to the identifiers "466322" and "466312".

The tokeniser also determines how calling parameters are matched to declared parameters. In textual languages, such a match is made by the relative position of the parameters in parameter lists. As discussed in chapter five, GILT uses a spatial matching technique. The tokeniser therefore performs a simple sorting and searching technique to generate "parameter ordering tokens" for Channel Ports, Channel Stubs, Passed Parameter Icons and Declared Parameter Icons. This could be performed by the parser with tokens giving the position of a diagram components on the screen, but the operation is better performed in C, simply because C is faster and more efficient at sorting a list of numbers than is Prolog.

7.5.3 Hierarchy tokens

The hierarchical nature of GILT diagrams is expressed using hierarchy tokens. Two different sorts of hierarchy tokens, "parent-child tokens" and "instance-definition tokens", exist. Both link diagram component nodes to definition nodes via their associated identifiers.

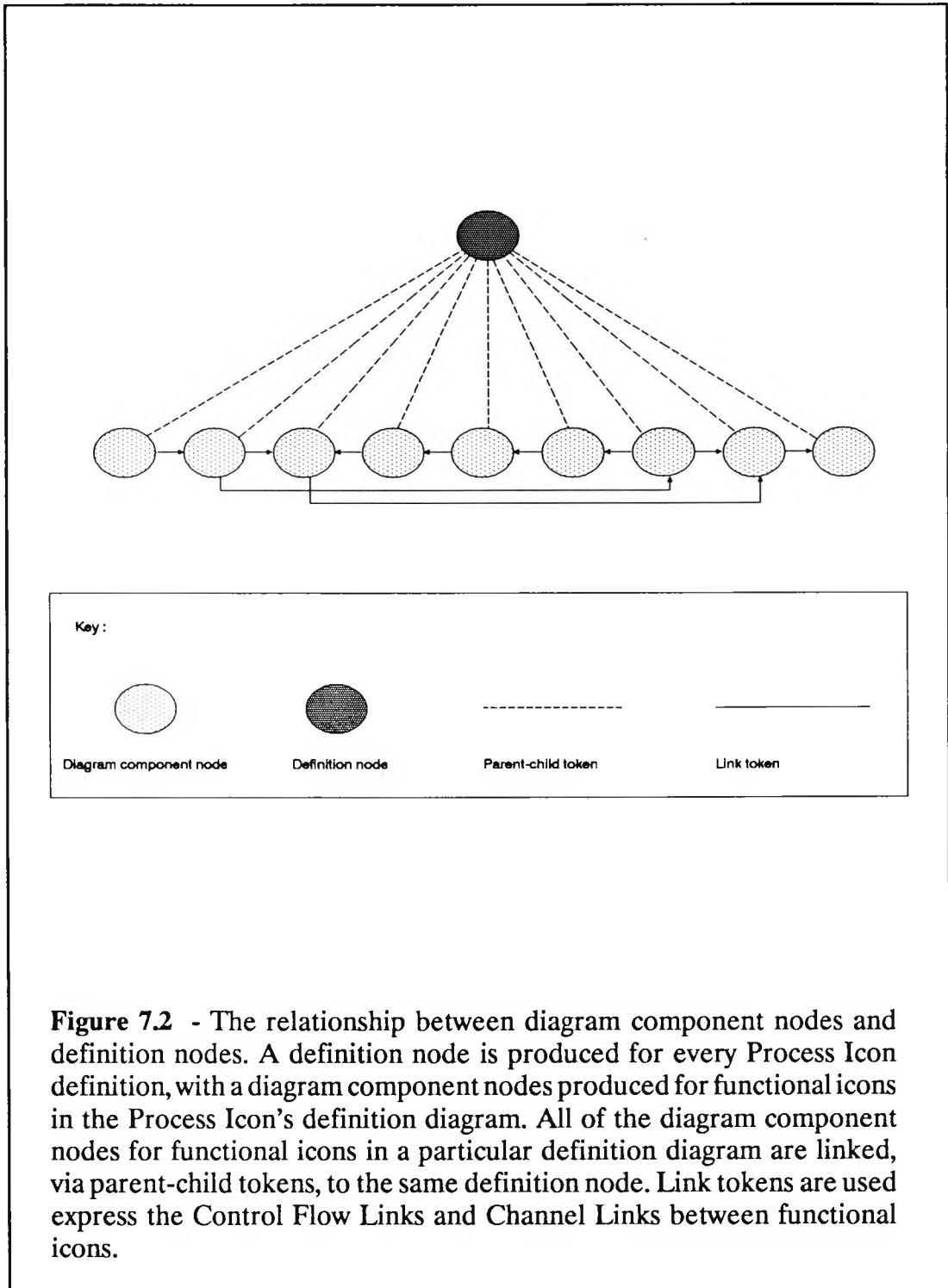
Parent-child tokens are used to link all of the diagram component nodes which represent functional icons in a particular definition diagram to a single definition node representing a Process Icon definition. All of the tokens for a particular diagram are linked to those of the Process Icon definition whose functionality they define. The structures produced in this way may be thought of as single rooted trees, like the one shown in figure 7.2. One such tree is produced for every graphical Process Icon, so the output from the tokeniser consists, at least in part, of a number of single rooted trees. The leaves of the trees are diagram component nodes, while the roots of the trees are definition nodes. All of the trees are disjoint, so that no node is in two separate trees.

Example

If the Guard Icon represented by the tokens in the previous example (section 7.5.2) were contained in the definition diagram of the earlier Process Icon definition, the following parent- child token would be output :

```
child_of_process_definition(471248, 450656) .
```

The instance-definition hierarchy token produced for every Process Icon instance links the tokens describing the instance to the tokens describing the Process Icon Definition which defines its functionality. Each instance-definition token references a diagram component identifier and a definition identifier, linking a diagram component node to a definition node. Instance- definition tokens may be thought of as combining the single rooted trees mentioned earlier into a more complex structure (actually a dependency tree) consisting of trees wired leaf to root, as shown in figure 7.3.



Example

If an instance of the Process Icon definition in the earlier example of section 7.5.1 (assigned the Definition Identifier "450656") were assigned the Instance Identifier "485544", the following Instance-definition token would be output :

```
instance_of_definition(485544, 450656).
```

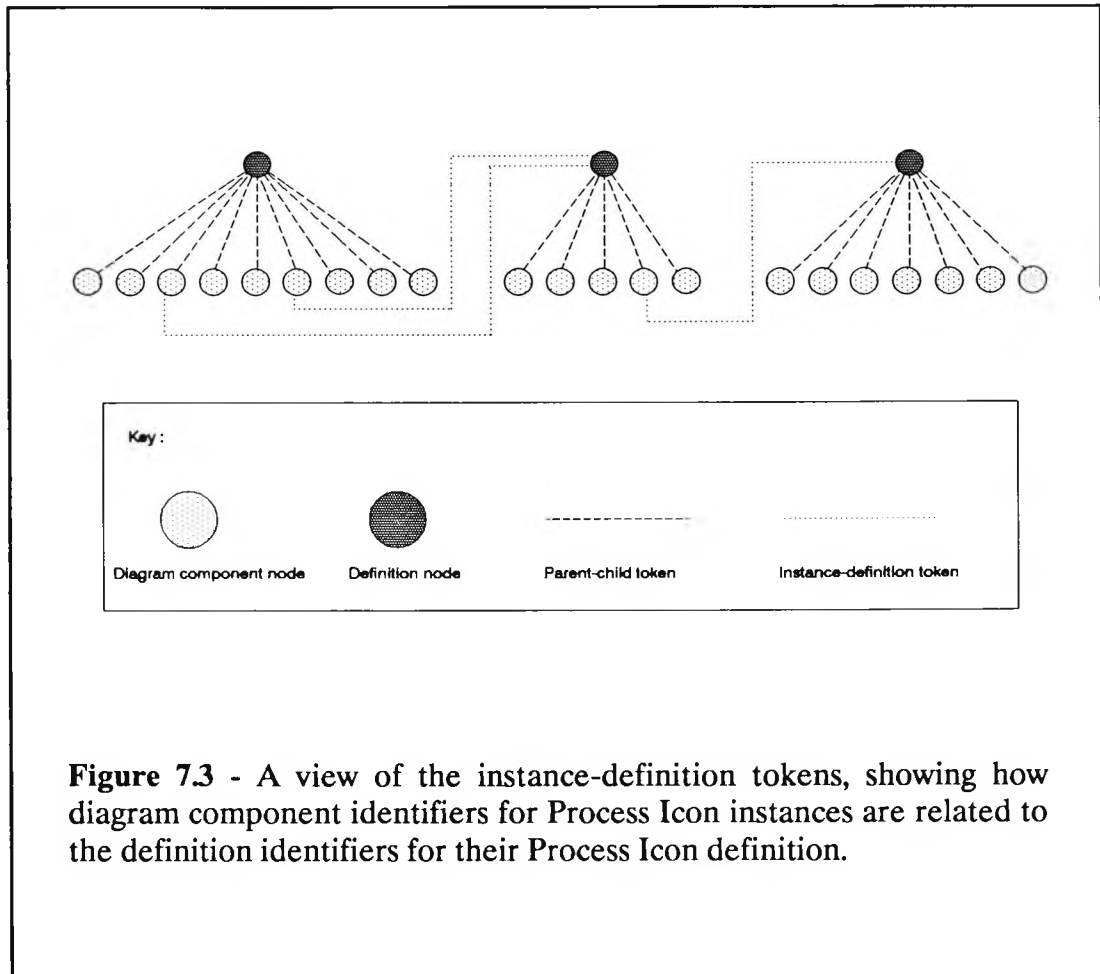


Figure 7.3 - A view of the instance-definition tokens, showing how diagram component identifiers for Process Icon instances are related to the definition identifiers for their Process Icon definition.

7.5.4 Example output from the tokeniser

An example of the tokeniser output is shown in figure 7.4 (next page). The figure contains the Prolog clauses produced by the tokeniser for a graphically defined double "buffer.2" Process Icon and a textually defined single buffer processes "buffer.1". The definition diagram for the double buffer process is shown in figure 7.5. The numbers of figures 7.4 and 7.5 show the relationship between the tokeniser output and the definition diagram.

7.6 Transfer of tokens between the tokeniser and the parser

In conventional compilers the tokeniser (or lexer) is usually implemented as a co-routine of the parser which supplies tokens on request. GILT's compiler differs from this approach by producing an intermediate file containing a set of tokenised Process Icon definitions and tokenised definition diagrams. The file is written by the tokeniser, and read in by the compiler. This was done primarily for practical reasons, because implementation of a more conventional system would have been difficult due to the limited interface between C and Prolog. It also allowed easy examination of the output from the tokeniser.

```
process_definition(450656).
process_definition_name(450656, 'buffer.2').
compile_as_procedure(450656).

/* 1 */
child_of_process_definition(467608, 450656).
    control_in_stub(467608).
    control_connect(467608,497008).

/* 2 */
child_of_process_definition(470816, 450656).
    control_out_stub(470816).

/* 3 */
child_of_process_definition(474024, 450656).
    channel_input_stub(474024).
    parameter_order(474024,0).

/* 4 */
child_of_process_definition(477232, 450656).
    channel_output_stub(477232).
    parameter_order(477232,0).

/* 5 */
child_of_process_definition(480440, 450656).
    process_instance_node(480440).
    instance_of_definition(480440, 503752).
    control_connect(480440,500040).

/* 6 */
child_of_process_definition(480441,450656).
    channel_output_port(480441).
    parameter_order(480441, 0).
    channel_connect(480440,480441).

/* 7 */
child_of_process_definition(480450,450656).
    channel_input_port(480450).
    parameter_order(480450, 0).
    channel_connect(480450, 480440).
```

Figure 7.4 - Output from the tokeniser for the double buffer example, the definition diagram of which is shown in figure 7.5. Indents are used to show which tokens form part of the same node. The numeric labels (e.g. "/* 1 */") shown are not part of the tokeniser output and have been inserted for use in relating the output to the definition diagram of figure 7.5. Continued over the next two pages.

```
/* 8 */
child_of_process_definition(484176, 450656).
  process_instance_node(484176).
  instance_of_definition(484176, 503752).
  control_connect(484176,500040).

/* 9 */
child_of_process_definition(484177,450656).
channel_output_port(484177).
parameter_order(484177, 0).
channel_connect(484176,484177).

/* 10 */
child_of_process_definition(484186,450656).
channel_input_port(484186).
parameter_order(484186, 0).
channel_connect(484186, 484176).

/* 11 */
child_of_process_definition(487912, 450656).
chancon_instance(487912).
channel_name(487912,'aaaaa').
channel_connect(480441,487912).
channel_connect(487912,484186).

/* 12 */
child_of_process_definition(490944, 450656).
chancon_instance(490944).
channel_name(490944,'baaaa').
channel_connect(484177,490944).
channel_connect(490944,477232).

/* 13 */
child_of_process_definition(493976, 450656).
chancon_instance(493976).
channel_name(493976,'caaaa').
channel_connect(474024,493976).
channel_connect(493976,480450).

/* 14 */
child_of_process_definition(497008, 450656).
control_split_join_node(497008).
control_connect(497008,484176).
control_connect(497008,480440).
```

Figure 7.4 - Output from the tokeniser for the double buffer example, continued.

```

/* 15 */
child_of_process_definition(500040, 450656).
  control_split_join_node(500040).
  control_connect(500040, 470816).

process_definition(503752).
process_definition_name(503752, 'buffer.1').
compile_as_procedure(503752).
text_filename(503752,
'/home/cb538/gilt/GLTAAa00622').
declared_input_channels_list(503752, ['in']).
declared_output_channels_list(503752, ['out']).
declared_variables_list(503752, []).
    
```

Figure 7.4 - Output from the tokeniser for the double buffer example, continued.

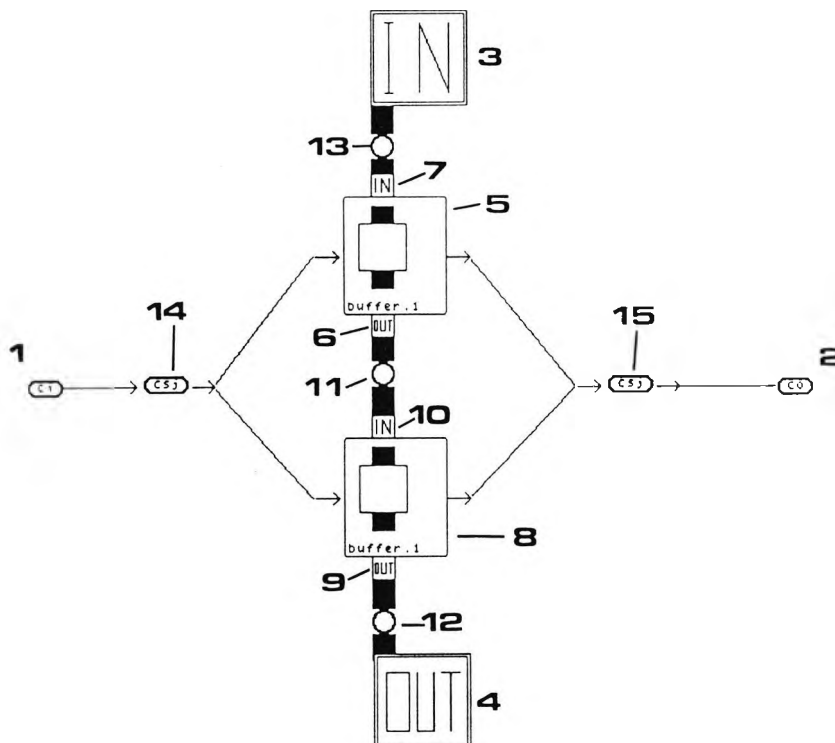


Figure 7.5 - A definition diagram for the double buffer example, labeled with numeric labels for reference to figure 7.4.

7.7 Parsing

The parsing of GILT diagrams poses considerably different problems to the parsing of textual languages, with the two dimensional nature of the diagrams and their corresponding definition in graph grammars precluding the use of the conventional tools and techniques which have been developed over a long time span for textual languages.

As discussed in chapter five, GILT diagrams may be thought of as containing three sets of information. The first set describes GILT's inter-process communications structures. The second set describes passed and declared non-channel parameters and the declared variables which might potentially be used in them. The third set is concerned with the control flow structures which specify the relative order of execution of processes, but have connection points for the channel connections. This set is defined by the base grammar of chapter five. The parsing of GILT diagrams may thus be considered as a three phase task. The first phase is concerned with the channel structures. The second phase deals with parsing non-channel parameter structures and declared variables. The third phase deals with the parsing of the remainder of the diagrams - the flow of control between Process Icons.

A parse tree is produced only for the constructs in GILT's base grammar because it would be difficult for a code generation routine to make use of the multiple parse trees which could be produced for the different aspects of the diagrams. Instead of producing explicit parse trees for the communications constructs and for the non-channel parameters, the results from the parses are distributed amongst different Prolog lists which are associated with identifiers. Lists are associated with definition identifiers (for Process Icon declared and local parameters) and with diagram component identifiers (for passed parameters).

The parsing of diagrams rests on the use of graph reductions. Graph reductions are used as a mechanism for parsing the diagram in a similar manner to the way in which textual "reductions" are used by, for example, shift-reduce textual language parsers. The following discussion therefore commences with a discussion of the graph reductions and is followed by two sections on the parsing of specific aspects of GILT diagrams. The first of the two sections deals with the parsing of communications structures, non-channel parameter structures and local variable definitions. The second deals with the parsing of the control flow structures in the base grammar.

7.7.1 Graph reductions and the parsing of GILT diagrams

In its most basic form, a graph reduction rule consists of three parts :

- (1) A graph which is to be removed (the "delete graph").

(2) A graph which is to be inserted (the "insert graph"). The insert graph is a less complex graph than the delete graph.

(3) A prescription for the way in which the insert graph is to be attached to the old graph - in essence a set of connection conditions.

Each graph reduction rule may be viewed as a production in a graph language which reduces, as opposed to increases, the complexity of the graph to which it is applied.

Any graph generated by a context free graph language may be parsed using a set of graph reduction rules repetitively applied (Della-Vigna and Ghezzi, 1978). In such cases, the insert graph is a single node, the "insert node", with an associated graph reduction created from a production in the grammar. One graph reduction rule is created for each rewriting on the right side of a production. The delete graph is formed by the right side(s) of a production in the grammar. The insert node is formed by the left side of a production. Each right side graph in a production has a pair of distinguished input and output nodes ("I" and "O"). These nodes form the attachment conditions for the insert node into the old graph, with any arcs incoming to the "I" node or outgoing from the "O" node reassigned to connect to or from the insert node.

A simple algorithm for parsing context free graph languages, which is similar to a conventional bottom up backtracking algorithm, may be obtained by repetitively cycling through all the graph reduction rules created for a particular grammar until no more reductions can be made. If a single node with a value or label equivalent to that of the start symbol of the context free grammar exists after all possible reductions have been applied, then the parsed graph is in the set of graphs defined by the graph language. Otherwise the graph is not within the set of graphs defined by the grammar. The algorithm may be applied to the parsing of GILT diagrams. Instead of generating one very large parse tree in complete form, which would be time consuming due to the time complexity of the algorithm, the algorithm is applied to the generation of large quantities of smaller parse trees which may be combined to form the larger tree. The parse trees generated are stored in the Prolog database as lists associated with identifiers, in a similar manner to the earlier parameter lists.

Graph reductions are encoded as Prolog goals in "reduction routines". Some reduction routines consist of a number of clauses which match tokens representing a complete delete graph, then replace them with appropriate new ones. Other routines progressively match, delete, and replace tokens representing parts of delete graphs until a complete graph has been identified and replaced. Reduction routines of this type are used for some grammar productions which have recursive elements, for example the production for the non-terminal "DECLARED_PARAMS" (figure 5.4e). Prolog's automatic backtracking mechanism is used to locate the tokens matching a particular delete graph, and allows a very natural expression of the delete graph, the insert graph and the connection conditions.

Reduction routines are applied successively over the separate definition diagrams by the use of extra clauses in delete routines. The extra clauses ensure that all of the tokens matched in a parse refer to diagram component identifiers related via parent-child tokens to the same definition identifier, and hence are all part of the same definition diagram. This minimises the amount of backtracking done by the Prolog system in its search for predicates satisfying reduction rules. It should be noted that the reductions made over the separated definition diagrams are not dependent on each other in any way, so many reduction routines could potentially be applied simultaneously in a parallel manner. Parallel execution of many reductions simultaneously is discussed in chapter 8.

Prolog's extra logical "retract" clause is used to remove tokens in the database representing parts of the delete graph and its external connections. Similarly, the extra-logical "assert" clause is used to add tokens representing the insert node or the parse trees and lists generated for particular structures. The database is thus maintained and modified by the reduction rules.

7.7.2 Parsing communications structures, non-channel parameter structures and local variable definitions

The parsing processes for communications structures, non-channel parameter structures and local variables are simple ones. Tokens for structures or local variable declarations are recognised using goals which match clauses in Prolog's database. Goals are repeatedly executed until they fail, indicating that no more reductions may be performed. Further goals remove tokens giving, for example, the relative order of parameters and the contents of their text areas. New tokens inserted into the database hold lists of parameters, variable declarations and channel declarations.

Five lists are produced for every definition identifier from tokens matched in the reduction routines. Each list contains textual parameters or declarations of one type from the text areas in a particular definition diagram. All the lists for the same definition diagram are associated with the same definition identifier, which references all of the instance nodes (and hence tokens) making up the diagram. In cases where no legal parameter structure can be found the list is empty. A "declared input channels list" for every definition identifier contains a list of channel names, one for each Channel Input Stub in the definition diagram, while a "declared output channels list" performs a similar purpose for the Channel Output Stubs. Channel names for stubs are obtained from the "channel_name" tokens associated with diagram component identifiers representing the Channel Connector Icons to which the Stubs are connected. The names of all Channel Connector Icons which are not connected to Stubs (and hence do not form part of a Process Icon's input parameters) are placed into a "local channels list". All of the stubs connected to the same Channel Connector (and hence part of the same communications construct) are assigned the same name, again obtained from the "channel_name" token associated with the instance identifier produced for each Channel Connector Icon. A "declared input variables list" contains a list of declared variable parameters obtained from the "dvar_name" tokens associated

with diagram component identifiers representing Declared Variable Icons. Similarly, a "local variables" list is generated to hold all of the definitions of local variables. The list contains the contents of the text areas forming a part of Variable Declaration Icons.

Figure 7.6 shows Prolog facts containing lists generated by the parser for the definition node with the definition identifier "450656" of figure 7.4. The declared input variables list is empty, as the Process Icon with the definition diagram of figure 7.5 has no declared input variables. Likewise, the local variables list is empty, due to the fact that no graphically defined variables are included in the diagram.

```
declared_input_channels_list(450656,[caaaa]).  
declared_output_channels_list(450656,[baaaa]).  
declared_variables_list(450656,[]).  
local_channels_list(450656,[aaaaa]).  
local_variables_list(450656,[]).
```

Figure 7.6 - Prolog facts containing lists generated by the parser for the definition node with definition identifier "450656" of figure 7.4.

A corresponding set of lists are generated for each instance identifier representing a Process Icon instance. A "passed input channels list" holds names for all of the Channel Input Ports of a Process Icon instance, while a "passed output channels list" serves the same purpose for Channel Input Ports. Like the earlier Stub names, all of the Channel Ports attached to the same Channel Connector item are assigned the same name from the "channel_name" token associated with each instance identifier representing a Channel Connector Icon. A "passed input variables list" holds all of the passed non-channel parameters for the Process Icon instance. Again, each list contains parameters, generated from tokens associated with diagram component identifiers, but the parameters are the calling parameters for Process Icon instances.

Figure 7.7 shows Prolog facts containing lists generated by the parser for the two Process Icon instances in the definition diagram of figure 7.5. The lists are shown associated with the identifiers generated by the tokeniser for the Process Icon instances. The passed variables lists are empty, as no variables are passed to either of the Process Icon instances.

As mentioned earlier, declared input channel lists, declared output channel lists, and declared parameter lists are associated with definition identifiers, while passed input channel lists, passed output channels lists and passed parameter lists are associated with instance identifiers. There exists a one-to-one relationship between these lists such that:

```
passed_output_channels_list(480440,[aaaaa]).
passed_input_channels_list(480440,[caaaa]).
passed_variables_list(480440,[ ]).

passed_output_channels_list(484176,[baaaa]).
passed_input_channels_list(484176,[aaaaa]).
passed_variables_list(484176,[ ]).
```

Figure 7.7 - Prolog facts containing lists generated for the two Process Icon instances in the definition diagram of figure 7.5.

declared input channel lists	<-->	passed input channel lists
declared output channel lists	<-->	passed output channel lists
declared parameter lists	<-->	passed parameter lists.

(<--> indicates a relationship)

Declared parameters for definition identifiers and passed parameters for instance identifiers therefore match each other. The "declared" lists provide declared parameters for a process implementing a Process Icon definition, while "passed" lists define calling parameters for an instance of such a Process Icon.

7.7.3 Parsing the control flow graphs

After the reductions of the previous section have been performed, the database contains a number of definition identifiers with associated declared and passed parameter lists as described above. All of the original instance tokens relating to communications structures and to passed, declared, or local variable definitions have been removed, with only those representing Control Flow Links and the non-channel parts of functional icons being left. As an example of the state of the database on entry to the control flow parsing routine, the internal database from the two buffer examples of figures 7.4 and 7.5 is reproduced in figure 7.8.

The parsing process for the remaining structures consists of performing multiple reductions over the graphs represented by the remaining tokens, checking that the control flow structures which they describe are valid ones, and producing a parse tree for them.

The control flow parser routine repetitively cycles through a number of graph reductions which reduce all nine of GILT's base grammar constructs which are described in section 5.2.5.1. A parse tree is built in a novel way which involves

```
process_definition(450656).
process_definition_name(450656, 'buffer.2').
compile_as_procedure(450656).
declared_input_channels_list(450656,[caaaa]).
declared_output_channels_list(450656,[baaaa]).
declared_variables_list(450656,[]).
local_channels_list(450656,[aaaaa]).
local_variables_list(450656,[]).

child_of_process_definition(480440, 450656).
    process_instance_node(480440).
    instance_of_definition(480440, 503752).
    passed_output_channels_list(480440,[aaaaa]).
    passed_input_channels_list(480440,[caaaa]).
    passed_variables_list(480440,[]).
    control_connect(480440,500040).

child_of_process_definition(484176, 450656).
    process_instance_node(484176).
    instance_of_definition(484176, 503752).
    passed_output_channels_list(484176,[baaaa]).
    passed_input_channels_list(484176,[aaaaa]).
    passed_variables_list(484176,[]).
    control_connect(484176,500040).

child_of_process_definition(497008, 450656).
    control_split_join_node(497008).
    control_connect(497008,484176).
    control_connect(497008,480440).

child_of_process_definition(500040, 450656).
    control_split_join_node(500040).
    control_connect(500040,470816).

process_definition(503752).
process_definition_name(503752, 'buffer.1').
compile_as_procedure(503752).
text_filename(503752,'/home/cb538/gilt/GLTAAa00622'
).
declared_input_channels_list(503752, ['in']).
declared_output_channels_list(503752, ['out']).
declared_variables_list(503752, []).
```

Figure 7.8 - The state of the Prolog database on entry to the part of the parser concerned with parsing the control flow structures. Indents are used to indicate tokens forming part of the same node.

propagation of nested lists between nodes being reduced and newly created ones.

Initially, the parser assigns a simple list (the "description") with a single element to each diagram component node representing a Process Icon. The single element is the identifier for the instance identifier with which it is associated. When a reduction for a particular construct is performed, the descriptions of the components of the construct are concatenated to form a new list. This list is treated as a list element and added to the end of a new list which contains an atom reflecting the nature of the reduction made (a "construct keyword") so that the first element of the list specifies the type of the construct of which the following arguments are a part. Finally the new list is assigned as the description of the newly created insert node. An example showing how parse trees are constructed using this method is shown in figure 7.9. Initially (a), the control flow graph consists of five nodes. Three nodes with diagram component identifiers "B" and "C" and "D" are wired into a parallel construct via the two control split join nodes "CSJ". This construct is wired into a sequential construct with nodes having the diagram component identifiers "A" and "E". After a graph reduction for GILT's parallel construct is carried out, the graph becomes the one shown in (b). The new node has the description "[PAR, B, C, D]". Carrying out a sequential reduction yields a single node with the description "[SEQ, A, [PAR, B, C, D], E]". This nested list gives a syntax tree for the graph, as shown in Figure 7.10. At each "level" the list consists of a construct keyword (here "SEQ" or "PAR") which indicates the type of a construct, followed by a number of parameters defining the components of the construct. Each parameter is in turn a list, defined in the same way, until the parameters are diagram component identifiers. Each instance identifier forms a leaf on the syntax tree, while non-leaf nodes correspond to keywords describing the type of the construct of which its descendant nodes are components. Constructs with no component constructs, such as GILT's skip and stop constructs are leaf nodes on parse trees.

The process of generating the parse tree is simplified by the associativity of GILT and Occam parallel constructs. The associativity of constructs means that, for example, in Occam,

```
PAR
  A
  PAR
    B
    C
```

is directly equivalent to

```
PAR
  A
  B
  C
```

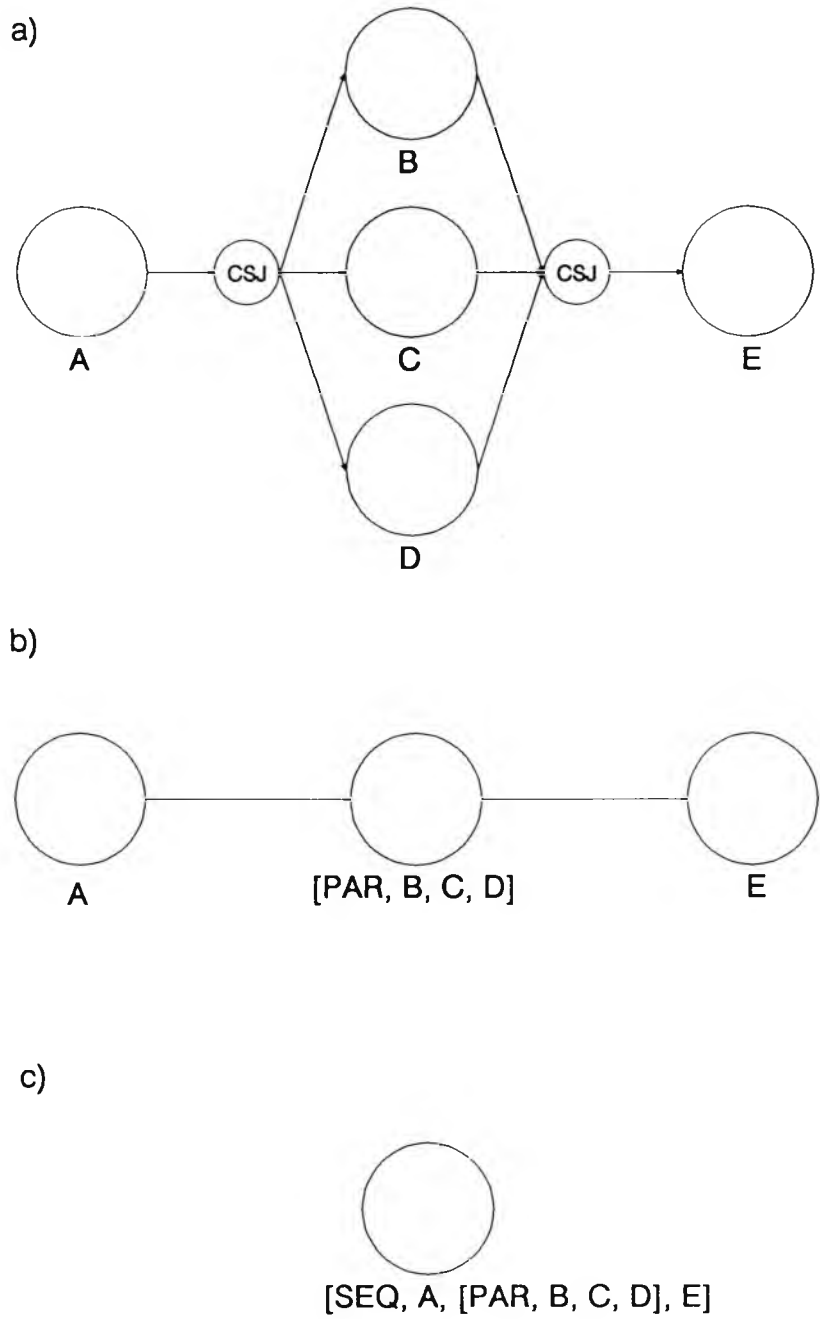
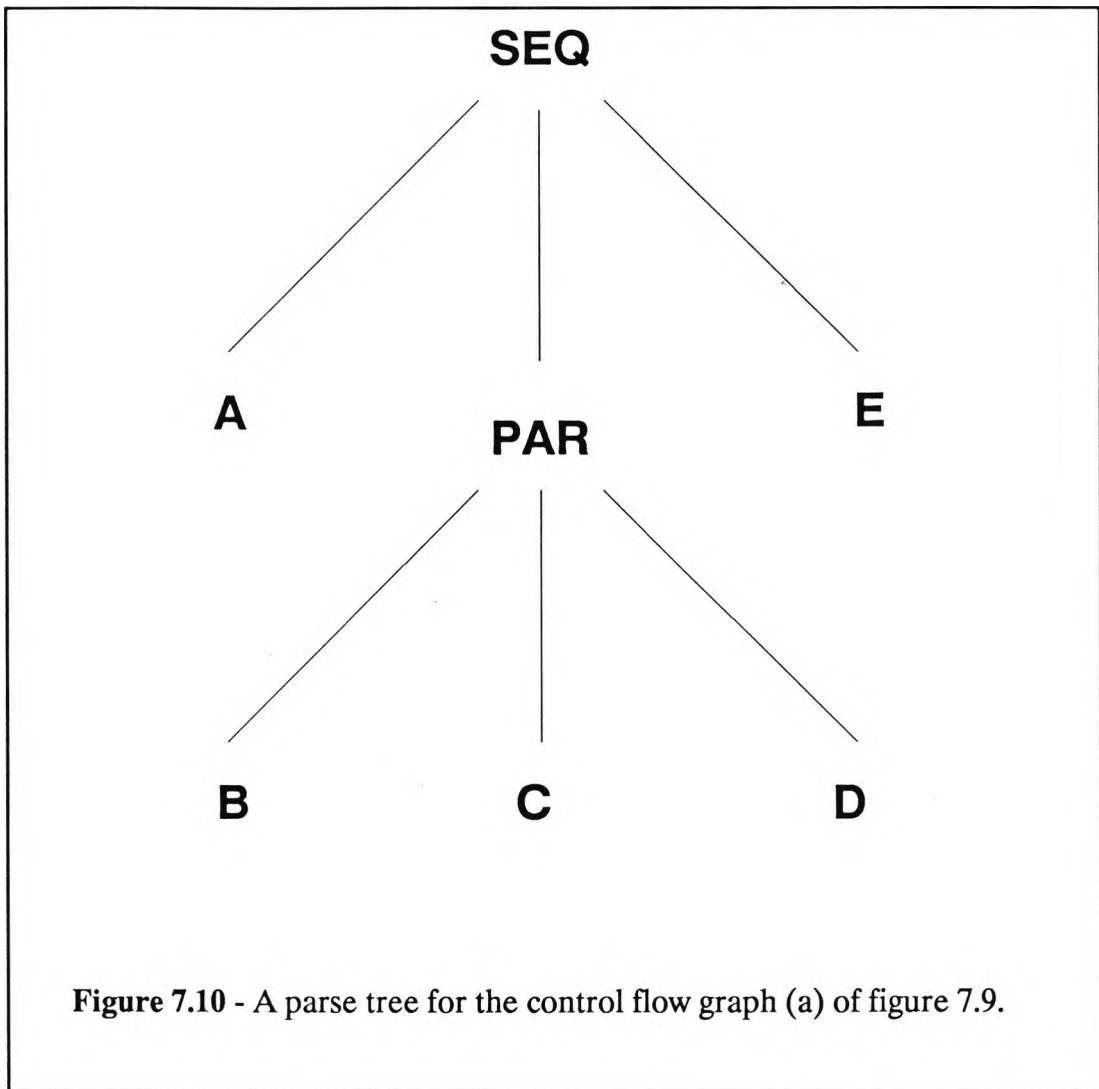


Figure 7.9 - The reduction of a control flow graph showing how a parse tree for the graph is constructed. Unlabelled nodes correspond to Process Icons, while those labelled "CSJ" correspond to Control Split Join Icons.



Similar laws exist for SEQ and ALT constructs in Occam and their equivalents in GILT. An example of GILT's associativity is shown by the two equivalent parallel constructs in figure 7.11. Associativity extends to syntax and parse trees, with the two syntax trees shown in figure 7.12 also being equivalent.

Because of the associativity of GILT's constructs, parsing routines may be allowed to generate parse trees for n-way parallel constructs like the one shown in figure 7.13. This considerably simplifies the graph reduction process by allowing the implementation of an n-way parallel structure recogniser with two very simple reduction rules. The first, context dependent, reduction reduces two nodes connected in parallel between two control split join nodes to a single node connected between the two control split join nodes. The second reduction reduces a single process connected between two control split join nodes to a single node. Providing that as many of the first reductions as can be performed are carried out before the second reduction, parallel structure may be correctly recognised by the reductions.

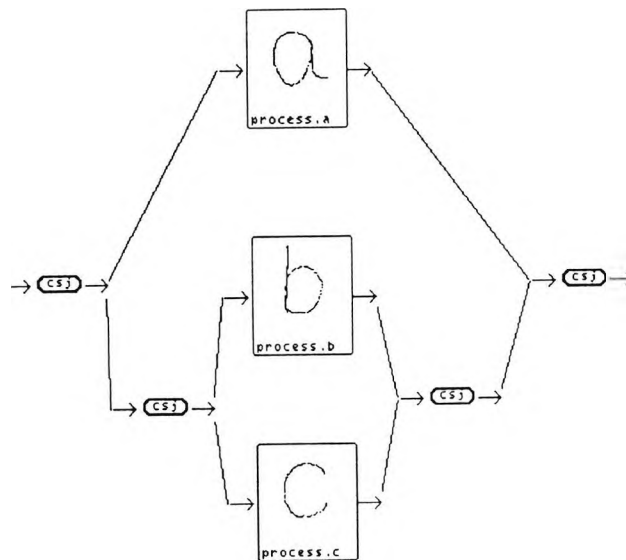
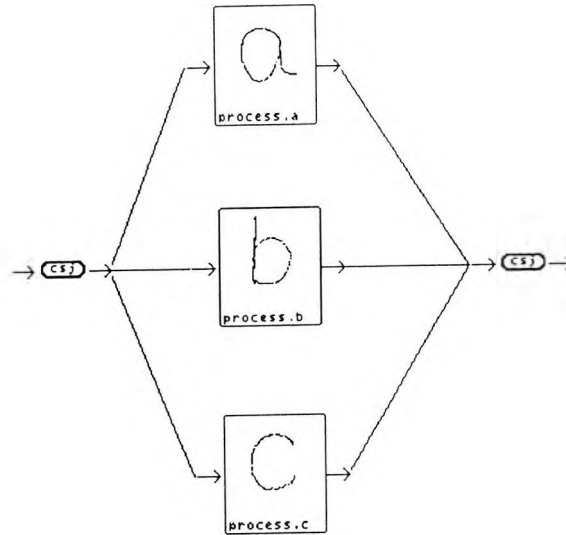


Figure 7.11 - Equivalent parallel constructs in GILT.

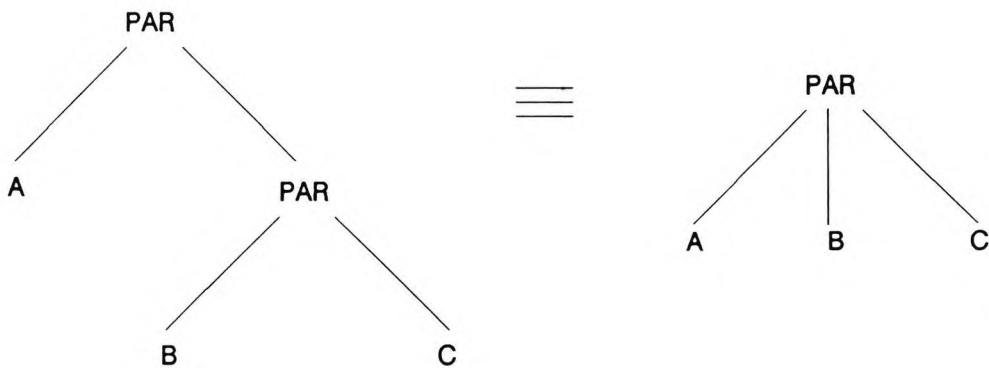


Figure 7.12 - Two equivalent parse trees.

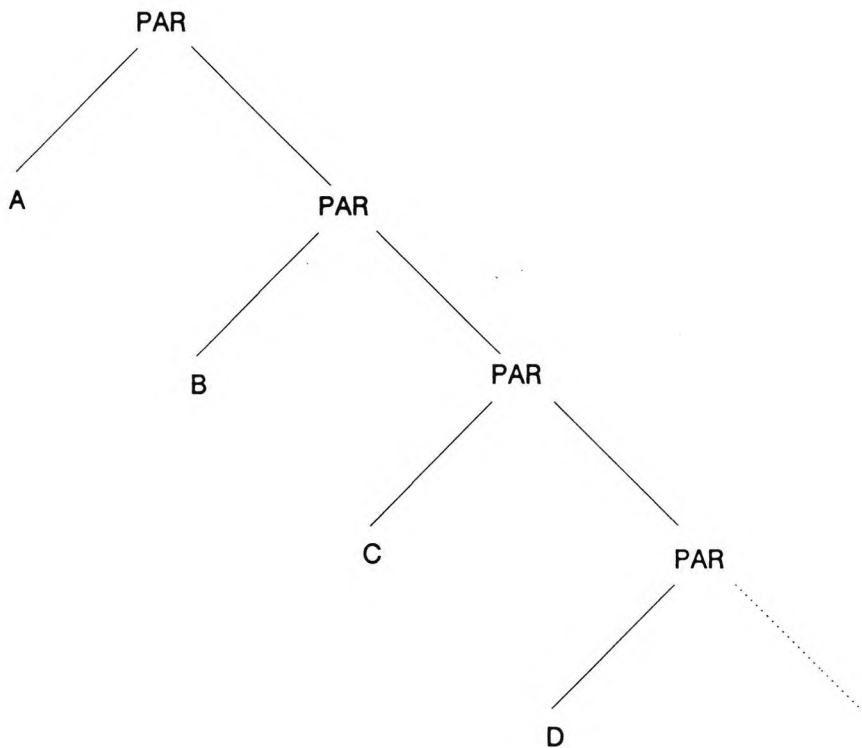


Figure 7.13 - A parse tree for an n-way parallel construct, making use of construct associativity.

Similar reductions are used for most of GILT's control flow constructs, resulting in a smaller parser than would otherwise be the case. An example parse of the earlier control flow graph of figure 7.9(a) using the methods discussed above is shown in figure 7.14.

Most of the control flow reductions look for patterns of "process_instance_node" token sets connected with other instance nodes and produce "process_instance_node" token sets. This saves a reduction transforming each "process_instance_node" token set into a "construct" token set. An example of a reduction using this scheme (actually a parallel reduction) is shown in figure 7.15.

The routine of figure 7.15 is called with most of its parameters "anonymous variables", as below :

```
parallel(Def_node, _, _, _, _).
```

The routine implements the first of the two parallel reductions mentioned earlier and used in figure 7.14. The anonymous variable ("_") is a variable which may be instantiated by Prolog to match any value.

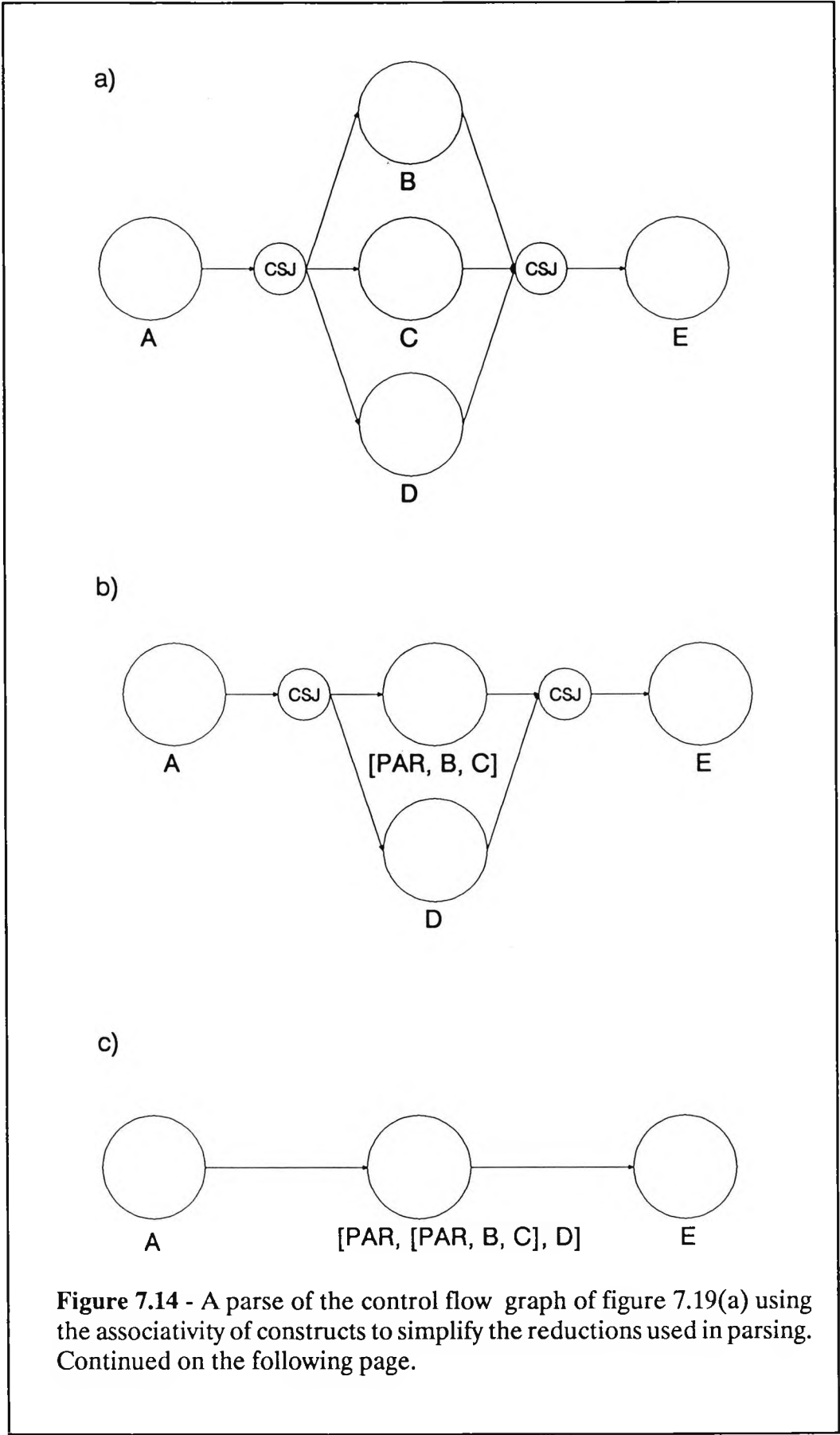
The routine may be divided into five sections A-E, as shown in figure 7.15 with Prolog's comment character "%":

Section A consists of a set of clauses which match tokens in the database representing the parallel structure that the reduction is designed to find. The "child_of_process_definition" clauses are used to restrict the amount of backtracking done by Prolog while it attempts to find suitable tokens, thus ensuring that Prolog does not waste excessive amounts of time searching the database for reductions which have no chance of being made. The clauses perform a similar function to restrictions in conventional bottom-up limited backtrack compilers which limit the number of symbols which may be examined in the search for a particular pattern. The last two clauses in the section ensure that the routine does not match erroneous structures consisting of the same nodes wired into peculiar patterns.

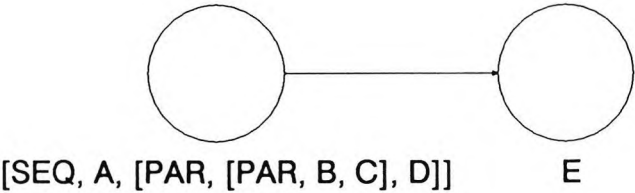
Section B obtains the "description" of each node, for usage in parse tree generation, as discussed earlier.

Section C creates a new node wired between the two control flow split-join nodes found by section A. The node is assigned a new unique identifier from the global variable "N", with the tokens describing the node created using the extra logical database manipulation predicate "assert". The description of the newly created node is created from the two descriptions obtained in section C and the string "PAR".

Section D removes the tokens from the database using the extra logical "retract" clause.



d)



e)

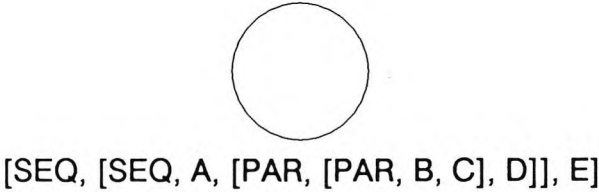


Figure 7.14 - continued.

```
parallel(Def_node,A,B,C,D) :-
    % section A
    child_of_process_definition(A, Def_node),
    control_split_join_node(A),
    child_of_process_definition(B, Def_node),
    control_split_join_node(B),
    child_of_process_definition(C, Def_node),
    process_instance_node(C),
    child_of_process_definition(D, Def_node),
    process_instance_node(D),
    control_connect(A,C),
    control_connect(C,B),
    control_connect(A,D),
    control_connect(D,B),
    A =\= B,
    C =\= D,

    % section B
    description(C,DC),
    description(D,DD),

    % section C
    new_node_number(N),
    assert(child_of_process_definition(N, Def_node)),
    assert(process_instance_node(N)),
    assert(description(N,['PAR',DC,DD])),
    assert(control_connect(A,N)),
    assert(control_connect(N,B)),

    % section D
    retract(control_connect(A,C)),
    retract(control_connect(A,D)),
    retract(control_connect(C,B)),
    retract(control_connect(D,B)),
    retract(process_instance_node(C)),
    retract(process_instance_node(D)),
    retract(description(C,DC)),
    retract(description(D,DD)),

    % section E
    retract(new_node_number(N)),
    N1 is N + 1,
    assert(new_node_number(N1)),
    set_change_flag.
```

Figure 7.15 - An example graph reduction routine, encoded in Prolog, which recognises parts of parallel constructs.

Finally, section E increments the global number "N" used to create identifiers for new token sets, and uses the predicate "set_change_flag" to set a flag which indicates that changes have been made to the database.

Similar reductions are used for the remainder of GILT's constructs. Some of the reductions are more complex, some less. The reductions are repetitively applied to Prolog's internal database until no more reductions may be performed. If the reduction of a particular diagram has been successful (i.e. the pattern of tokens emitted by the tokeniser was a legal graph), all of the diagram's constructs are reduced to a single instance node. The description of the node which, in the case of a successful reduction, contains a parse tree for the diagram is associated with a relevant definition identifier. The definition identifier also references the lists produced by earlier sections of the parser, and so provides a complete description of a Process Icon definition in a form that can be used by the code generator.

7.7.4 Error checking

After all the possible reductions have been performed on the graphs stored in Prolog's database, a test can be applied to see if there are any structures which have not been reduced remaining in the database. This is performed by Prolog goals which attempt to find specific tokens in the database. If any such tokens are located, their unique identifier and a suitable error message is output. The identifier may be selected using the mouse and used to locate a diagram component for the identifier using the user interface to the compiler (section 7.9.1).

7.8 Code Generation

Code generation for GILT is a much simpler task than in conventional compilers. No attention needs to be paid to usually important issues like storage allocation, register usage, etc., all of which is handled by the Occam compiler. The code generator simply translates the parse tree and lists produced by the parser into an Occam program, taking care that the construct spacing required by the Occam compiler is correct.

The error handler discussed above ensures that the routines to produce Occam code from the lists and parse tree produced by the parser are only called if the graphs have been fully reduced, so that of the original tokens, only the definition-instance tokens remain. All of the other tokens produced by the tokeniser have been removed and replaced with lists associated with instance and definition identifiers. As described in sections 7.7.2 the lists hold partial parse trees (one parse tree for each definition diagram), lists of local channels or variables, and passed and declared parameters.

An example of the information held in Prolog's internal database on entry to the code generation routines is shown in figure 7.16.

```
process_definition(450656).
process_definition_name(450656, 'buffer.2').
compile_as_procedure(450656).
declared_input_channels_list(450656,[caaaa]).
declared_output_channels_list(450656,[baaaa]).
declared_variables_list(450656,[]).
local_channels_list(450656,[aaaaa]).
local_variables_list(450656,[]).
description(450656, [PAR, 480440, 484176]).

instance_of_definition(480440, 503752).
passed_output_channels_list(480440,[aaaaa]).
passed_input_channels_list(480440,[caaaa]).
passed_variables_list(480440,[]).

instance_of_definition(484176, 503752).
passed_output_channels_list(484176,[baaaa]).
passed_input_channels_list(484176,[aaaaa]).
passed_variables_list(484176,[]).

process_definition(503752).
process_definition_name(503752, 'buffer.1').
compile_as_procedure(503752).
text_filename(503752,
'/home/cb538/gilt/GLTAAa00622').
declared_input_channels_list(503752, ['in']).
declared_output_channels_list(503752, ['out']).
declared_variables_list(503752, []).
```

Figure 7.16 - State of the Prolog internal database on entry to the code generation routine.

The code generator produces an Occam procedure definition for each definition identifier (e.g.. "450656" and "503752" above) having a "compile_as_procedure" token, thus creating a separate Occam procedure for every procedural Process Icon. The text contained in the "process_definition_name" token provides a name for every Occam procedure definition produced by the code generator. This allows easy examination of the code produced by the compiler and makes possible the manual insertion of Occam's mapping directives (e.g. "PLACED PAR"), as it is easy to identify which Occam procedures have been generated from which Process Icon definitions. Calls to the named procedures are inserted into generated code as required.

Occam procedure definitions are not output for definition identifiers with "compile_as_code_insert" tokens. Instead code for the definition identifiers, which represent non-procedural Process Icons, is output as a non-procedural Occam process where required.

The output of a procedure definition for a given definition identifier may be viewed as a three stage process :

- 1) Output of a header for the procedure.
- 2) Output of a set of local channel and variable declarations.
- 3) Output of the body of the procedure definition.

The header for the procedure consists of a procedure name (discussed earlier) and a set of associated declared parameters, enclosed by brackets. The declared parameters are supplied by the declared output channels list, the declared input channels list and the declared variables list which are associated with the definition identifier.

Local variable definitions and the names of channels contained in the local variables list and the local channels list associated with the definition identifier respectively are used to generate local channel and variable definitions immediately after the procedure header.

For definition identifiers describing textual Process Icon definitions, the body of the procedure is the contents of the icon's text file. For definition identifiers describing graphical Process Icon definitions, the body of the procedure is produced from the description list associated with the definition identifier. A simple breadth first tree walking algorithm is used to produce Occam from the parse tree represented by the list. Each keyword in the list identifies a construct and determines how its associated parameters are output. Instance identifiers in the lists (for example "480440" and "484176" above) generate references to procedures or code inserts, while construct keywords are used to produce Occam reserved words. The decision to output a procedure reference or a code insert for an instance identifier is taken depending on the procedural or the non-procedural nature of the definition identifier which is related to the instance identifier in question via a instance-definition token. Thus, procedure calls are generated for instances of procedural process icons, while code inserts are generated by instances of non-procedural process icons.

Procedure calls are extremely simple to output. They consist of the name of the procedure being called, followed by a bracket enclosed list of parameters, obtained from the passed parameter lists associated with the instance identifier being output.

Example

The call shown below is generated for the instance identifier "480440" contained in the earlier figures :

```
buffer.1(aaaaa, caaaa)
```

For non-procedural instance identifiers an "equivalence table" is output, followed by variable and channel declarations local to the process and by code for the body of process. The equivalence table is produced using Occam's "IS" object abbreviation facility and relates calling parameters (in the passed lists associated with an instance identifier) to declared parameters (in the declared lists associated with a definition identifier) by renaming.

The code for the body of the process produced is obtained by a breadth first tree walk down the parse tree of the definition identifier related to the instance node identifier, as above. Local channel and variable declarations are output in the same fashion as for the earlier procedure definition.

Example

If Process Icon "buffer.1" of the buffer example were modified so that it were non-procedural instead of procedural, the token "compile_as_procedure(503752)." of figure 7.4 would be replaced by the token "compile_as_code_insert(503752)". Correspondingly, the code generated for instances of the Process Icon would reflect this change, so that the output "buffer.1(aaaaa, caaaa)" of the previous example would be replaced by :

```
in IS caaaa :
out IS aaaaa :
INT x :
WHILE TRUE
  SEQ
    in ? x
    out ! x
```

Procedures are output in an order determined by instance-definition tokens, which define a form of dependency tree. After all of the component procedures have been output, a reference to a top-level procedure is output in the form required for compilation of the Occam code output as an "EXE" under the TDS, if possible. An "EXE" is an Occam program suitable for execution on a single Transputer (specifically the host Transputer), and consists of a single sequential process, with no declared channels or parameters, unlike the example two buffer example used throughout this chapter :

```
SEQ
  toplevel()
:
```

GILT's compiler will compile programs which do not conform to this model (and have Channel Stubs or declared non-channel parameters), but will not produce a reference to the top-level procedure, as shown above. Instead, a warning message is output.

The toplevel procedure is determined by a parameter for the goal `"compile_gilt_to_occam(d)"`, which compiles the set of tokens loaded into the Prolog database. The parameter `"d"` specifies a definition identifier which is to be compiled as the top-level procedure. The goal is automatically executed by a specialised interface (section 7.9) so that users of GILT need have no knowledge of the compiler's internal workings. After completion of the code generation process started by the goal, a message is produced giving the name of the file that the compiler has placed output in. The filename is obtained from the name of the top-level Process Icon indicated by the definition identifier `"d"` with an added suffix `".occ"`, used by GILT to indicate files containing Occam code.

Figure 7.17 gives an example of the output from the compiler produced for the two buffer example. If the Process Icon `"buffer.1"` were non-procedural, as described above, the output in figure 7.18 would be obtained. The statement `"#USE userio"` at the beginning of each code segment is generated by the compiler from the contents of a library specifier file `"gilt_libs.occ"` which must be contained in every directory in which the GILT compiler is used. The file contains a list (possibly empty) of Occam `"#USE"` library directives. Procedures defined in the libraries may be used in the Occam code of textual Process Icons.

7.9 Interaction with the compiler

Though the Prolog system provided a very good environment for development of the compiler some knowledge of Prolog was required in order to use it. Therefore, a user interface to the compiler was produced.

The "compiler window" consists of two connected windows, as shown in figure 7.19. The upper "control panel" contains three buttons for interacting with the lower part of the window and with the rest of the GILT system. The "compiler sub-window" is a Sunview tty sub-window which runs a Prolog shell for the compiler. The compiler sub-window displays messages from the compiler and will accept textual commands given as Prolog goals. Tty sub-windows are components of the Sunview system designed to allow the easy creation of graphical interfaces to existing applications. Once a tty sub-window has been created, a shell may be executed in the window asynchronously to the application which started it. Routines are provided in libraries to insert text into the input buffer of the shell and extract text from its output buffer. The compiler user interface makes use of these facilities by starting a Prolog shell in the compiler sub-window, loading the compiler into it and interacting with it via Prolog goals sent to the shell's input buffer.

```
#USE userio
PROC buffer.1(CHAN OF INT in,CHAN OF INT out)
  INT x :

  WHILE TRUE
    SEQ
      in ? x
      out ! x
  :
PROC buffer.2(CHAN OF INT caaaa,CHAN OF INT baaaa)
  CHAN OF INT aaaaa :
  PAR
    buffer.1(caaaa,aaaaa)
    buffer.1(aaaaa,baaaa)
  :
```

Figure 7.17 - Output from the compiler for the two buffer example. The first procedure was output for the Process Icon definition named "buffer.1" with definition identifier "503752". The second procedure was output for the Process Icon definition named "buffer.2" with definition identifier "450656".

```
#USE userio
PROC buffer.2(CHAN OF INT caaaa,CHAN OF INT baaaa)
  CHAN OF INT aaaaa :
  PAR
    in IS caaaa :
    out IS aaaaa :
    INT x :
    WHILE TRUE
      SEQ
        in ? x
        out ! x
    in IS aaaaa :
    out IS baaaa :
    INT x :
    WHILE TRUE
      SEQ
        in ? x
        out ! x
  :
```

Figure 7.18 - Output from the compiler for the two buffer example modified so that the code produced for non-procedural processes may be examined. No procedure was output for the Process Icon definition named "buffer.1" with definition identifier "503752", as it was compiled as inline code.

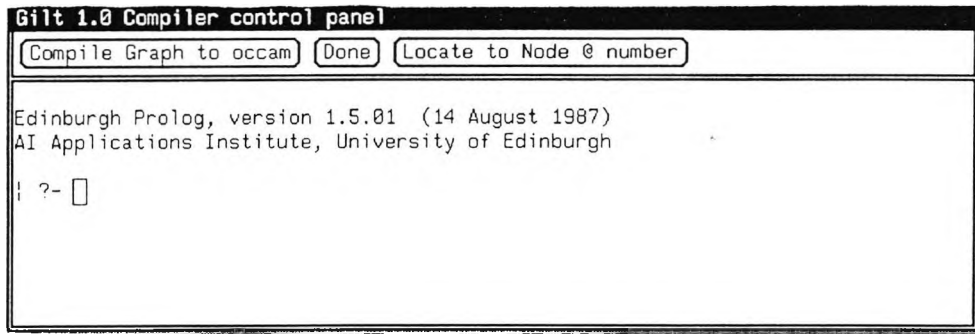


Figure 7.19 The complete compiler window shown in its initial state. The boot up message from the Edinburgh Prolog system used may be seen in the tty sub-window.

7.9.1 The control panel

Three control panel buttons are provided in the control panel : "Compile Graph to Occam", "Locate to node @ number" and "Done".

Compile Graph to Occam (button)

The control panel button "Compile Graph to Occam" has an associated event handling procedure which sends commands to the compiler as Prolog goals via the library routines mentioned above. Every button push initiates the following sequence of events :

- 1) The tokeniser is called and a file containing a tokenised version of the current GILT program produced.
- 2) Commands are sent to the Prolog shell to load the intermediate file produced by the tokeniser. If the button push is the first one during the editing session, a command is sent to load the compiler routines. On subsequent pushes, a command is sent to reset the compiler to a known state.
- 3) The goal "compile_gilt_to_occam(d)." is sent to the compiler instructing it to produce code for the Process Icon definition currently being edited as the top-level procedure, where "d" is the definition identifier of the relevant Process Icon definition.

The compiler then starts compilation, producing appropriate messages as and when required.

Locate to node @ Number (button)

This button is used to relate error messages produced by the compiler back to the definition diagrams displayed by the editor, providing a visual error location facility. The system makes use of Sunview's "selection service", which allows text to be highlighted by using the mouse. Any text highlighted in such a manner when the "Locate to node @ Number" button is pressed is checked to see if it is a legal instance identifier. If it is, the identifier is used to locate the part of the functional icon forming an erroneous structure and display it in reverse video. Figures 7.20 and 7.21 illustrates the use of this feature, which has proved a very good debugging tool. The format of the error messages produced by the compiler as described in more detail in section 7.9.2.

Done (button)

The done button closes the compiler window, removing it from view. The compiler window may be displayed again (or for the first time) using the "Compiler Window" button in the diagram editor control panel.

7.9.2 The compiler sub-window

The compiler sub-window runs a Prolog shell, as discussed earlier. Users of the system may interact with the compiler by typing Prolog goals, or use the facilities provided by the control panel. The main use of the panel is in the display of error and advisory messages.

Error messages all have similar styles, giving a textual error message and a numeric instance identifier, as shown by the example of figure 7.20. The instance identifier may be used to locate to the erroneous parts of the diagrams. The error messages from the compiler may be regarded as giving messages similar to those provided by conventional compilers, which commonly consist of a textual part (the error message) and a line number. The messages are self explanatory, and will not be dealt with in depth here.

Advisory messages are used by the compiler to reassure users about operations in progress, provide information on compiler output files and for warnings. Messages are produced for most phases in the compilation. A final message is output to indicate successful compilation and to indicate the production of an output file containing Occam code. Figure 7.22 shows such a message. Warning messages are similar to error messages, but give information of a less fatal nature, for example specifying that the toplevel procedure call for an EXE has not been output as the Process Icon definition chosen is not suitable (section 7.8). A warning message is shown in figure 7.23.

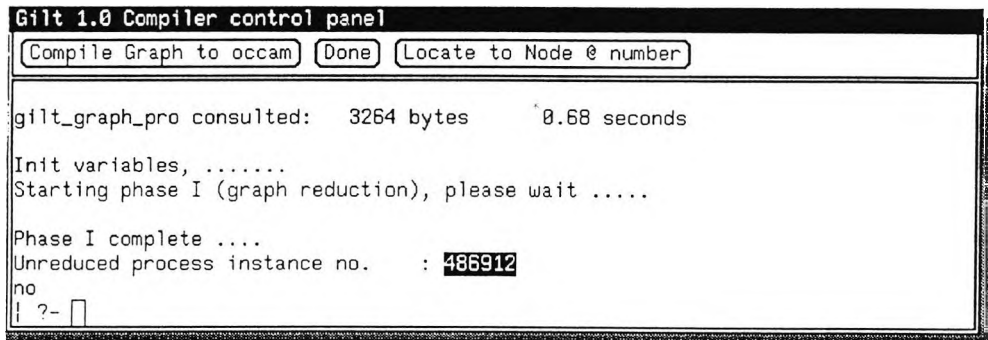


Figure 7.20 - The compiler window showing an error message from the compiler indicating a node which has not been reduced, and thus an erroneous diagram. An instance identifier has been highlighted using the mouse so that the error location facility may be used.

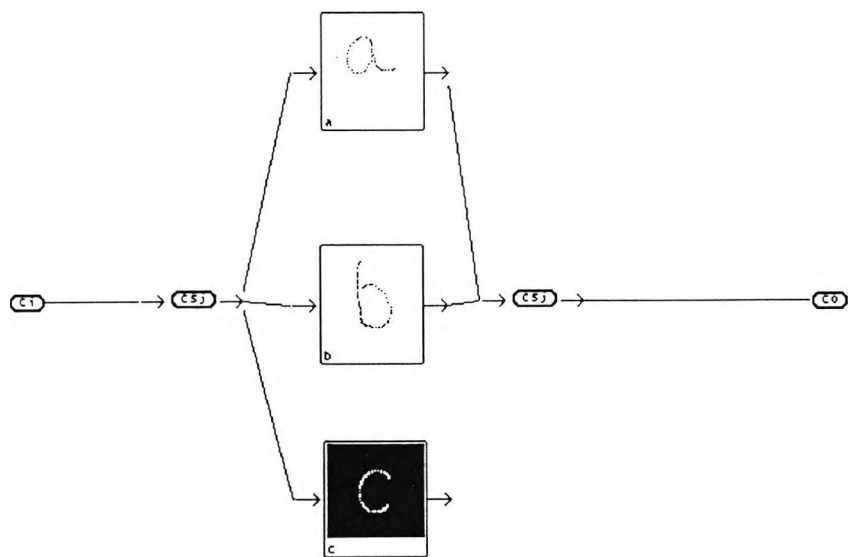


Figure 7.21 - A view of an erroneous diagram showing a functional icon highlighted by the use of the error location facility. The icon highlighted is the one indicated by the error message of figure 7.20.

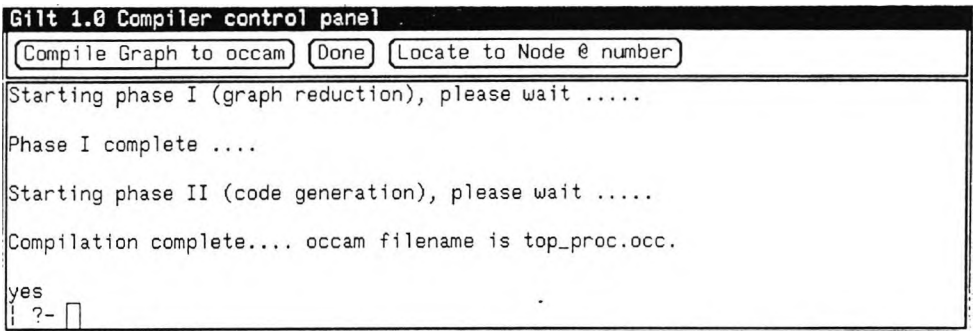


Figure 7.22 - Successful compilation message giving the filename containing the compiler output.

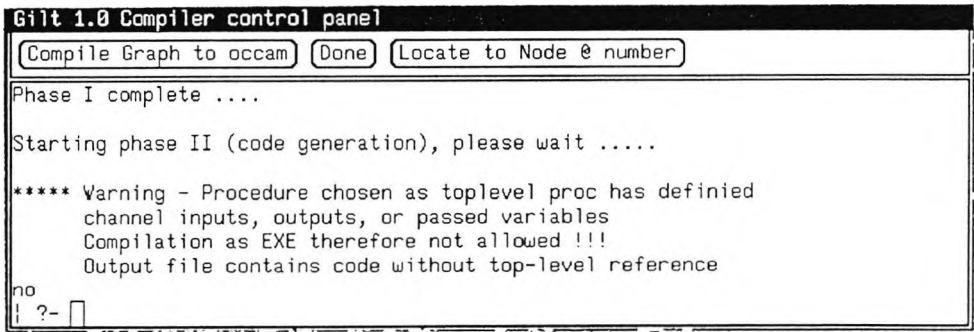


Figure 7.23 - An example warning message.

Results, conclusions and suggestions for future work

8.0 Introduction

The results of the work presented in this thesis are mainly methods concerned with the construction of parallel visual programming systems and are closely allied with the conclusions derived from the work. Hence, results and conclusions are discussed together in this chapter. Discussion of the results and conclusions is followed by a section providing suggestions for future work.

8.1 Results and conclusions

In the development of any new programming language, it is difficult to obtain metrics on the usability of the language without extensive user testing. At the final evaluation, language longevity is perhaps the ultimate test. As the objective of the thesis was primarily in the investigative domain, user testing of the prototype system was not carried out. The results and conclusions of this section therefore concentrate on giving an empirical analysis of what has been achieved during the course of the research. The research has involved the construction of a fully working prototype visual programming system, with which a number of examples have been coded. Several new principles and techniques for the implementation of visual programming systems for concurrent computation have also been developed in the course of the work. Specific results are discussed in separate (following) sections, and are followed by a very brief summary.

8.1.1 A visual programming system for parallel computation

GILT adopts a new approach to writing concurrent programs for von-Neumann style multiprocessors. Imperative programs are represented using a paradigm which includes both text and graphics. The graphical aspects of GILT include visualisations of control flow and inter-process communication, which have not previously been combined into a unified model. The combination of the visualisations of control flow and communications with a mixed textual and graphical programming paradigm allows the graphical representation of concurrent programming structures not visualised in previous systems, for

example alternative structures. It also allows visual programming techniques to be used for very high level overviews and for lower level detail. Very low level detail, representing operations on abstract objects like variables, is expressed using conventional textual code. Most previous visual programming systems have used visual programming exclusively for high level overviews or for low level detail. GILT is unique in allowing visual programming to be used over a wide range of levels of abstraction, from a very high level to a lower one and offers a choice of structural overviews of programs. It does not however impose restrictions on where and when visual or textual programming techniques should be used. Instead GILT allows users to make such choices. Users may determine the most appropriate medium (graphics or text) for the expression of the algorithm concerned.

GILT is also the first visual programming system to make a distinction between procedural and non-procedural hierarchical structures. This difference is considered important, as it allows programs to have a hierarchical structure independent of the one conventionally imposed by procedural abstraction. It facilitates visual abstraction and aids the efficient management of screen area, a feature which has been noted (Myers, 1988) as lacking from many previous systems.

It is the author's opinion that a visual programming system based on the ideas presented in this thesis is an appropriate technology for programming current and near future parallel computers, and offers the potential of creating an easy to use interface to concurrency. In addition, visual programming techniques are applicable to the "next" generation of personal computers based on styluses, handwriting recognition and flat "writable" screens. It is not difficult to imagine the production of a GILT like programming system making use of handwriting recognition for the input of textual code while relying, like GILT, on line drawings and the selection of pre-defined icons for graphical functions.

8.1.2 The use of graph grammars for visual language syntax

Another major result of the work was the production of a new method of defining the syntax of visual languages. Context free graph grammars are simple and easy to use, yet offer sufficiently expressive qualities to describe visual languages, as is proven by the description of the GILT language in chapter five. Previous work has been based on the use of context dependent grammars, or has restricted itself to the use of context-free grammars for visually simple languages. The work presented in chapter four uses context free grammars, netting their associated desirable properties, yet still allows the expression of context-dependent information. It should be able to be used widely in visual languages which have different overlaid sets of information forming their visualisation.

GILT is one of the few visual programming languages in existence which has a well defined syntax, and the only one (to the author's knowledge) which has a syntax expressed using a combination of graph grammars and text grammars.

8.1.3 Construction of visual programming systems from standard user interface components

Previous visual programming systems have been implemented using highly specialised environments. The construction of the prototype GILT system has shown that it is possible to implement a visual language using very general user interface components. The use of such user interface components allows the development time of a visual language to be reduced considerably over what would otherwise be the case, yet still allows a highly language specific interface to be created, which is not the case with syntax directed diagram editors.

8.1.4 A compiler for a visual language

GILT is one of a very few compiled parallel visual programming languages. The Prolog implementation of the compiler's parser is sufficient for the parsing of small to medium scale programs in GILT, and provides a very clear expression of the ideas involved in the compiler. Parsing large programs does however require a long time period due to the time complexity of the parsing algorithm used. The running time of almost all backtracking algorithms is, at worst, $O(e^n)$, where n is the number of "objects" being parsed. GILT's limited backtracking algorithm improves on this to a great extent. The number of comparisons made per definition diagram is strictly limited by an upper bound imposed by the editing system on the number of functional icons which may occur in any given diagram. The running time for the building of a parse tree and lists for each definition diagram is thus bounded so that the for a GILT program of n definition diagrams it is, at most, $O(n)$. However, as $O(1)$ is fairly high, so the compiler does run relatively slowly. For example, the compilation of the processor farm example of chapter five takes approximately eleven seconds. Suggestions for speeding up the compilation of GILT programs are included in section 8.2.

8.1.5 Summary of conclusions and results

The work presented in this thesis has contributed a new method of syntactic specification for visual languages i.e. the use of multiple context free graph grammars, produced a new visualisation for Occam style parallel programs incorporating a mixed control flow/channel paradigm and shown how standard user interface components may be used for producing fully functional visual programming systems within short time spans. A nontrivial visual programming system supporting the GILT language has been produced. The system includes far more support for parallel programming than have previous visual programming systems, though there is certainly room for improvement. The development of a compiler for GILT has demonstrated useful approaches to the compilation of visual languages and resulted in a usable prototype visual programming system which has been demonstrated to numerous people and has received universally favourable comments.

8.2 Suggestions for future work

The previous works discussed in the review section of the thesis and the body of the work presented in the preceding chapters strongly suggest that graphical program development tools have a role to play in the development of parallel systems. However, there is much outstanding work to do. At a very basic level, a study of user's reactions to graphical tools for the development of parallel software systems should be carried out to assess the strengths and weaknesses of graphical representations in parallel programming.

The suggestions for future work included in the following sections of this chapter are of two kinds. Some concern enhancements to the existing system which would allow it to be more easily and efficiently utilised while others indicate interesting directions for research on future graphical programming development tools designed to aid parallel programming.

8.2.1 Improvements and enhancements to the existing system

Although the prototype system is easy to use, a number of minor improvements would increase its functionality. The improvements are ordered in descending importance.

8.2.1.1 Checking of textual syntactic entities.

At present the GILT programming system performs no checking on the textual parts of GILT diagrams which are defined in GILT's syntax using extended BNF expressions. The textual parts of the diagrams are considered as correct syntactic entities and are passed straight through to the code generation section of the compiler. Thus it is possible to input, for example, incorrect Occam into a text window, have it passed through the compiler as part of a GILT program, and for the compiler to generate incorrect code because of the original erroneous text embedded in it. On further compilation of the output from the compiler, using the Transputer Development System, such errors are detected but relating them back to the original code is difficult. What is required is simply a number of independent lexers and parsers for all of the textual parts of the GILT language which produce appropriate error messages. No code generation need take place, but the error messages produced would be used for detection purposes and to ensure that incorrect text was not propagated into the Occam compiler. The construction of parsers for the textual parts of the GILT language was not undertaken due to time constraints, but it represents no substantial problem and could easily be completed using standard compiler generation tools in a few months.

8.2.1.2 Configuration of GILT programs

The compiler built for GILT produces unconfigured Occam, suitable for compilation for a single Transputer as an "EXE" under the TDS. This code can easily be configured manually for multiple transputers by inserting configuration statements into the output from the GILT compiler before it is input to the Occam compiler. There is no fundamental problem with modifying the code generation sections of the compiler to produce code directly for multiple transputers, but additional information would have to be supplied to generate the configuration information. In line with the rest of the project, a graphical "pick and place" mapping tool would be appropriate. The configuration process could be approached in a topdown, graphical, manner assigning groups of processes to "super groups" of processors, then carrying out stepwise refinement. Performance feedback in the form of coloured processor and channel activity coding like that used in GRAIL (Stepney, 1987) could be used to guide the assignment process.

8.2.1.3 More advanced editing facilities

The facilities provided by the program editor are sufficient for the creation of complex GILT programs but the addition of some extra features would be worthwhile. Enhancements to the editing system could include allowing different sized and shaped icons for the representation of processes, zooming (for viewing large diagrams), and multi-level diagram editing as well as relatively primitive features like a "snap to grid" function, block moving and copying. Channel Ports are at present constrained to the upper and lower surfaces of Process Icons, while Control Flow Ports must be attached to the left and right sides. This arrangement has worked well in most situations, but there are cases (such as regular grid process arrays and circular structures) where the ability to distribute ports around the outsides of icons in an arbitrary manner would be more appropriate. A "tidy" diagram feature, to move functional icons to appropriate positions on a grid would also be useful. Some work on automatic layout algorithms suitable for GILT's channel connections has already been performed (Kramer, Magee and Ng, 1989).

8.2.1.4 Support for channel protocols

GILT's visual channels (implemented using Channel Links, Channel Ports and Channel Stubs) support only simple integer protocols. Although not a major drawback for a prototype system, the expansion of the language to allow the use of more complex communications protocols would aid the construction of complex parallel programs. Occam's protocols are widely used by programmers, and could easily be included by the association of a pull-down menu with each Channel Stub, which would allow the "type" of the channel to be visually set. Alternatively, the visualisation user for Channel Stubs in the current implementation could be modified to include a text editing area for the definition of a protocol for the channel. Neither of these enhancements would involve significant work.

8.2.1.5 Parsing GILT diagrams

As mentioned earlier, GILT's compiler runs relatively slowly, which is a disadvantage for compiling large programs. Improvements to the run time for the compiler could be made in a number of different directions :

1) The parser could be recoded in an imperative, sequential language so that the overheads imposed by Prolog were removed.

2) The parser could be ported to a form of parallel logic language like Strand (Foster and Taylor, 1990) which provides similar facilities to

Prolog, but would allow the multiple graphs representing diagrams to be simultaneously reduced.

3) The parser could be re-implemented in GILT or in Occam so that multiple transputers could be applied to the reduction process. As the ratio of computation to the amount of data used by the computations is low, a processor farm would be an ideal solution. In such a system a farmer process would packet out separate tokenised diagrams to each worker process in the farm. The worker processes would perform reductions on the diagrams until no more could be performed, then send the lists and partial parse tree generated by the reductions back to the farmer process, which would continue dispatching diagram packets until no more were available. The processor farm example of chapter five gives an implementation for the main structure of a suitable farm. Such a system would even dynamically balance load on the Transputer system it ran on. As different amounts of computation are required for the generation of parses of complex diagrams and parses of simple ones, dynamic load balancing is desirable in a distributed parser for GILT diagrams. Processor utilisation would be kept extremely high in a processor farm parser.

Option (3) is considered the most efficient in terms of runtime overhead and also offers the elegant possibility of writing a compiler for GILT which is itself as a GILT program! A "new" GILT compiler would, of course, have to be compiled using the existing system.

Further improvements to any of the schemes above could be obtained by the use of a predictive graph parsing method, as described in (Kaul, 1982) and by only parsing the parts of diagrams which had been modified since the previous compilation. The production of a very fast parser for GILT diagrams would allow the development of a syntax directed editor for the language, or at the very least a highly interactive error detection system.

8.2.1.6 Support for the use of replicators and other constructs

Replicators, which allow the construction of regular arrays of processes using indexing variables, are one major feature of Occam not implemented in GILT. GILT does allow the creation of process arrays similar to those generated using replicators by the placing and wiring of components and by the use of hierarchy (for example the construction of an two element buffer from two one element buffers, the construction of a four element buffer from two two element buffers, and so on), but this approach is not appropriate for multi-dimensional regular structures due to the amount of connections required between the different levels of abstraction. Nonetheless, replicators are a major feature lacking from GILT. An interesting idea worthy of further investigation is the use of graph grammars (not necessarily context-free) to specify replicated structures. Productions in such a grammar could be visually input, and graph rewriting used to create the desired structure. Productions could even be generated using the techniques of programming by example (Myers, 1988), with a system deducing a regular connection pattern from a small sub-diagram. Alternatively, simple visual bracketing notations could be used, but this would become rapidly more complex for multi-dimensional structures.

The addition of further Occam-like constructs into GILT, for example "CASE" conditionals and multi-way "IF" constructs would require the addition of a few new functional icons and associated productions in the grammar of chapter five. Minor additions to the diagram editor would therefore need to be made.

8.2.1.7 Animated execution of GILT programs

The views that GILT provides of parallel programs are static ones. It is easy to see that animated execution of explicitly concurrent programs has advantages for debugging and for performance analysis. As a view of the program is already in existence (specifically a set of GILT diagrams), a graphical debugging or performance analysis tool does not need to generate its own visualisation, as has been the case with previous Transputer or Occam based systems. Animation of important concurrent concepts such as synchronised communications may help to reveal deadlocks and other related bugs. A variety of methods could be used to extract information from a transputer system, for example those discussed in (West, 1987) and (Zimmermann, 1988). A system allowing animated execution of GILT diagrams would be an invaluable tool for teaching, and this is considered to be a very worthwhile improvement.

8.2.2 Working towards future graphical program development tools

There are a number of interesting areas in the field of graphical program development tools for parallel systems worthy of investigation, some of which are described in the following sections.

8.2.2.1 Mixed paradigm approaches

GILT's mixed paradigm approach (control flow, channels, text and graphics) seems successful and could be an appropriate area for further research. Future systems could take the mixed graphics and text approach a stage further and allow conventional code to be mixed with graphics so that, for example, a textual for loop could enclose, or even replicate, a graphical construct.

8.2.2.2 Combined visual programming and program visualisation

The binary divide of programming systems into program visualisation systems and visual programming systems may not be necessary if text and graphics complement each other as well as they appear to do. It should be possible to parse in existing programs into a visual programming system and lay them out either by hand or automatically. Obviously such a "reverse engineering" approach does present significant difficulties, but these should not preclude research in this direction.

8.2.2.3 Visual mixed language programming

Mixed language programming is an area little investigated for concurrent programming. It is easy to imagine a mixed language system like GILT allowing the functionality of modules (Process Icons) to be described using different textual (or even graphical) languages with sets of compatible communications primitives. The techniques of visual programming could be used to specify the connections between the modules.

8.2.3.4 The use of colour

The use of colour in parallel programming is a good research area. Colour coding of processor activity has already been shown to be useful by a number of systems, but further research on other applications is needed. One possibility for the application of colour coding in GILT-like systems is to indicate potentially erroneous or incomplete structures using different hues. Another possibility is, as mentioned earlier, to use colour for the display of performance information.

8.2.3.5 Multidimensional tools

Graphical program development tools need not necessarily be limited to two dimensional representations as there are many common parallel programming structures for which three or higher dimensional representations are appropriate. Raytracing provides a mechanism for rendering startlingly realistic images of non-existent scenes, with "glass spheres" and "ball and stick" molecules being common

in such images. The techniques of virtual reality (Leib, 1990) provide methods for interacting with 3-D scenes and it is pleasant to image a graphical tool with a walk (fly?) though metaphor in which users wear special equipment (for example, suits and goggles with appropriate sensors) to enter a world of glass sphere processes connected in three (or more) dimensions via "pipes". Users could "carry" a kit of specialist tools for interacting with the environment, such as "channel wrenches". Interestingly, the mechanisms used in GILT's syntax and compiler would require little expansion to deal with such a "CSP world".

As a final concluding remark, graphical program development tools offer many exciting possibilities for the development of software systems for parallel computers, which are limited only by the imaginations of their designers.

Appendix 1

The syntax of Occam2

The syntax presented here is a version of that contained in (Jones and Goldsmith, 1988), to which the reader is directed for further information. It is included here to enable a comparison with the various syntactic expressions in the thesis, and assumes the definition of concepts such as a "name", "decimal", "hexadecimal", "real", in Occam's micro syntax. Occam's language structure is described by productions in a BNF modified to cope with the two dimensional syntax of occam. Non-terminal symbols are written as lower case, while terminals are enclosed by boxes and correspond to Occam reserved words and symbols. Vertical bars (|) are used to represent alternative parses for a class. Horizontal juxtaposition means that components appear after each other on a line. Vertical juxtaposition means that the components of a parse must appear above each other on separate lines. Two kinds of abbreviation are used for sequences of instances of the same kind :

parallel ::= **PAR**
 {process}

Implies that a parallel can have any number of instances of process in it, arranged above each other at the same indentation (here two spaces in from the indentation of the PAR.

fun.heading ::= {1 **base.type**} **FUNCTION** name {0 **fun.formals**}

Implies that fun.heading is laid out on a single line. It begins with a sequence of instances of base.type, at least one of them, separated by commas. Between the brackets there is a possibly empty sequence of instances of fun.formals separated by commas.

Data types

base.type ::= **BOOL** | **BYTE** | int.type | float.type

int.type ::= **INT16** | **INT32** | **INT64** | **INT**

float.type ::= **REAL32** | **REAL64**

data.type ::= base.type | [**expr**] data.type

Protocols

```
protocol ::= name | simple.protocol | anarchic

protocol.definition ::= sequential.definition |
                        discriminated.definition

simple.protocol ::= data.type | count.type[::[]]data.type

count.type ::= int.type | [BYTE]

sequential.definition ::=
                        PROTOCOL name [IS] sequential.protocol :

sequential.protocol ::= {1[;] simple.protocol}

discriminated.definition ::= [PROTOCOL] name
                             [CASE]
                             {tagged.protocol}
                             :

tagged.protocol ::= tag | tag;sequential.protocol

tag ::= name

anarchic ::= [ANY]
```

Literals or constants

```
literal ::= bool.literal | byte.literal | int.literal |
            float.literal

bool.literal ::= [TRUE] | [FALSE]

byte.literal ::= byte | byte([BYTE]) | integer([BYTE])

int.literal ::= integer | integer(int.type) |
                byte(int.type)

integer ::= decimal | hexadecimal

float.literal ::= real(float.type)
```

Arrays

```
table ::= [[{1[;]expr}] | string | name

part ::= table | part[expr] | [part [FROM] expr [FOR] expr ]
```

Specifications and scope

specification ::= declaration | abbreviation | definition

definition ::= protocol.definition | proc.definition |
fun.definition

Declaration

type ::= data.type | channel.type | port.type | timer.type

specifier ::= type | [[expr.option]]specifier

expr.option ::= expr | empty

declaration ::= type {1, n}name) :

variable ::= name | variable[[expr]] |
[variable FROM expr FOR expr]

channel.type ::= CHAN OF protocol | [[expr]] channel.type

channel ::= name | channel[[expr]] |
[[channel FROM expr FOR expr]]

port.type ::= PORT OF data.type | [[expr]]port.type

port ::= name | port [[expr]] |
[[port FROM expr FOR expr]]

timer.type ::= TIMER | [[expr]]timer.type

timer ::= name | timer[[expr]] |
timer FROM expr FOR expr]

Abbreviation

abbreviation ::= value.abbreviation |
object.abbreviation |
retyping

value.abbreviation ::= VAL specifier.option name IS expr :

specifier.option ::= specifier | empty

object.abbreviation ::= specifier.option name IS object :

object ::= variable | channel | port | timer

retyping ::= value.retyping | object.retyping

retyping ::= variable.retyping | object.retyping

value.retyping ::= **VAL** specifier name **RETYPE**s expr :
 object.retyping ::= specifier name **RETYPE**s variable :

Expressions, conversions, etc.

expr ::= rand | mon.op rand | rand rator rand | conversion |
 extremum

rand ::= literal | part | variable | **[**expr**]** | **[**value.proc**]** |
 fun.call

rator ::= number.op | modulo.op | bit.op | logic.op |
 shift.op | relate.op

mon.op ::= **-** | **MINUS** | **~** | **BITNOT** | **NOT** | **SIZE**

number.op ::= **+** | **-** | ***** | **/** | **** | **REM**

modulo.op ::= **PLUS** | **MINUS** | **TIMES**

logic.op ::= **AND** | **OR**

bit.op ::= **/** | **BITAND** | **** | **BITOR** | **>** | **<**

shift.op ::= **<<** | **>>**

relate.op ::= equality | inequality | after

equality ::= **=** | **<>**

inequality ::= **<** | **<=** | **>** | **>=**

after ::= **AFTER**

conversion ::= base.type rounding rand

rounding ::= empty | **ROUND** | **TRUNC**

extremum ::= **MOSTPOS** int.type | **MOSTNEG** int.type

value.proc ::= VALOF		specification
process		value.proc
RESULT expr.list		

Processes

process ::= atom | construct | discrimination | loop |
 instance | block

block ::= specification		allocation
process		process

$$\text{atom} ::= \boxed{\text{SKIP}} \mid \boxed{\text{STOP}} \mid \text{assignment} \mid \text{input} \mid \text{output}$$

Assignment

```
assignment ::= variable.list := expr.list
```

```
variable.list ::= {1   variable}
```

$$\text{expr.list} ::= (1 \mid \text{expr}) \mid (\text{value.proc} \mid \text{fun.call})$$

Communication

```
output ::= channel.output | port.output
```

```
channel.output ::= channel  source.list |
                  channel  tagged.source.list
```

$$\text{tagged.source.list} ::= \text{tag} \mid \text{tag} \boxed{;} \text{source.list}$$

```
source.list ::= {1 □ source}
```

$$\text{source} ::= \text{expr} \mid \text{expr} \boxed{::} \text{expr}$$

```
port.output ::= port ! expr
```

```
input ::= channel.input | port.input | timer.input | delay
```

```
channel.input ::= channel [?] target.list |
                  channel [?] CASE tagged.list
```

$$\text{tagged.list} ::= \text{tag} \mid \text{tag} \boxed{;} \text{target.list}$$

```
target.list ::= {1 ;target}
```

$$\text{target} ::= \text{variable} \mid \text{variable} \boxed{::} \text{variable}$$

```
port.input ::= port [?] variable
```

timer.input ::= timer [?] variable

$$\text{delay} ::= \text{timer } [?] \boxed{\text{AFTER}} \text{expr}$$

Constructed Processes

```
construct ::= sequence | conditional | parallel |
           alternation
```

$$\text{replicator} ::= \text{name} \mid \text{expr} \mid \text{FOR expr}$$

Sequential Process

sequence ::= $\boxed{\text{SEQ}}$ {process} | $\boxed{\text{SEQ}}$ replicator
process

Conditional process

conditional ::= $\boxed{\text{IF}}$ {choice} | $\boxed{\text{IF}}$ replicator
choice

choice ::= guarded.choice | conditional | specification
choice

guarded.choice ::= expr
process

Homogeneous choice

discrimination ::= selection | case.input

selection ::= $\boxed{\text{CASE}}$ expr
{option}

option ::= {1[expr]
process | $\boxed{\text{ELSE}}$
process | specification
option

case.input ::= channel [?] $\boxed{\text{CASE}}$
{variant}

variant ::= tagged.list | specification
process variant

Parallel Processes

parallel ::= $\boxed{\text{PAR}}$ {process} |
 $\boxed{\text{PAR}}$ replicator |
process placed.parallel | pri.parallel

placed.parallel ::= $\boxed{\text{PLACED}}$ $\boxed{\text{PAR}}$ {placement} |
 $\boxed{\text{PLACED}}$ $\boxed{\text{PAR}}$ replicator
placement

placement ::= singleton | placed.parallel

singleton ::= $\boxed{\text{PROCESSOR}}$ expr
process

allocation ::= $\boxed{\text{PLACE}}$ name $\boxed{\text{AT}}$ expr [:]

```

pri.parallel ::= [PRI] [PAR]
                {process}
                [PRI] [PAR] replicator
                process
    
```

Alternative Process

```

alternation ::= [ALT]
                {alternative}
                [ALT] replicator
                {alternative}
                pri.alternation
    
```

```

alternative ::= simple.alternative | alternation |
                specification
                alternative
    
```

```

simple.alternative ::= guarded.alternative |
                    case.alternative
    
```

```

guarded.alternative ::= guard
                    process
    
```

```

guard ::= [SKIP] | expr [&] [SKIP] | input | expr [&] input
    
```

```

case.alternative ::= channel [?] [CASE] | expr [&] channel [?] [CASE]
                    {variant} | {variant}
    
```

```

pri.alternative ::= [PRI] [ALT]
                    {alternative}
                    [PRI] [ALT] replicator
                    alternative
    
```

Unbounded Loops

```

loop ::= [WHILE] expr
        process
    
```

Procedure abstraction

```

proc.definition ::= proc.heading
                 process
                 [ ]
    
```

```

proc.heading ::= PROC name [(0 [ , proc.formals ] )]
    
```

```

proc.formals ::= specifier(1, name) | [VAL] specifier(1, name)
    
```

Function Abstraction

```
fun.definition ::= fun.heading IS expr.list : |  
                  fun.heading  
                  value.proc  
                  :
```

```
fun.heading ::= {1 : base.type} FUNCTION name (  
                  {0 : fun.formals )
```

```
fun.formals ::= VAL specifier {1 : name}
```

```
fun.call ::= name ( {0 : fun.actual )
```

```
fun.actual ::= expr
```

Appendix 2

Definitions of hierarchical graphs, graph grammars and legal GILT graphs

The syntax of the GILT language is based on "hierarchical valued graphs". Hierarchical graphs are in turn based on "valued graphs". Thus this definition begins with a definition of a valued graph, and proceeds with a definition of a hierarchical valued graph or H-graph. Finally, some conditions for legal GILT graphs are defined.

Definition - valued graph

A set of symbols is termed a "vocabulary". Assume that V_M and V_A are finite sets of distinct symbols. V_M and V_A are the sets of node values and arc labels respectively.

A valued graph G over V_M and V_A is a triple (N, L, E) where N is a finite set of nodes.

$L: N \rightarrow V_M$ (L , the node value function, defines the value of each node)

$E \rightarrow (N \times V_A \times N)$ (E - the arc set, defines the arcs of G and their labels).

If $(n, a, m) \in E$, then an arc exists from node n to node m with label a . If G is a graph, then N_G , L_G and E_G denote the node set, node value function and arc set of G respectively.

If V_M and V_A are finite sets of distinct symbols then the vocabulary $V^*(V_M, V_A) = \{G \mid G \text{ is a graph over } V_M, V_A\}$. In shorthand, where V_M and V_A are assumed it is written V^* and is informally the set of all graphs composed with nodes having values from V_M and arcs having labels from V_A , including the "empty graph".

The empty graph, denoted ϵ , has no nodes or edges. The notation V^+ denotes $V^* - \{\epsilon\}$.

Definition - hierarchical valued graph

An hierarchical valued graph or H-graph over V_M, V_A is defined as follows:

The vocabulary of a level-0 H-graph $H_0^*(V_M, V_A) = V_M$.

A level-1 H-graph is a valued graph over V_M and V_A , as defined previously :

$H_1^*(V_M, V_A) = V^*(V_M, V_A)$, the set of all level-1 H-graphs.

A level-k H-graph ($k \geq 1$) over V_M, V_A is a graph over

$\bigcup_{i=0}^{k-1} H_i^*(V_M, V_A)$ providing that V_A

has at least one node value in $H_{k-1}^*(V_M, V_A)$

$H_k^*(V_M, V_A)$ (in shorthand written H_k^*) = $\{X \mid X \text{ is a level-}k \text{ H-graph}\}$

$H^*(V_M, V_A) = \bigcup_{k=0}^{\infty} H_k^*$, the set of all H-graphs over V_M, V_A , in shorthand written H^* .

The notation H^+ denotes $H^* - \{\epsilon\}$.

Thus, a level k hierarchical valued graph is composed of nodes which have values which are level k-1 hierarchical valued graphs. A level 1 hierarchical valued graph is a simple valued graph.

Definition - H-graph grammar :

A "H-graph grammar" is a quintuple (V_t, V_n, V_a, S, P) where :

V_t is a finite set of "terminal" node labels (the "terminals"),

V_n is a finite set of "non-terminal" node labels (the "non-terminals"),

V_a is a finite set of arc labels (the "arcs"),

S , the "start symbol", is a distinguished member of V_n ,

and P is a set of "productions" s.t. each production is a quadruple (G, H, I, O) and written $G \rightarrow H, I, O$. The "left part" $G \in V_n$. The "right part" is H, I, O . In general $H \in H^*$. In our case we do not allow H to be the empty graph ϵ and hence $H \in H^+$. I and O are distinguished nodes in H termed the "input" and "output nodes" (or "gluing points") respectively.

Productions are used to derive graphs with node values in H^+ starting from a "host graph" containing only the start symbol S . During the derivation, the nodes of the host graph with values in V_n (e.g. an arbitrary node A valued B , with $B \in V_n$) are replaced by the right part of some production rewriting B , e.g. $B \rightarrow C, I, O$. Every

arc originally entering (exiting) the node B becomes an arc entering I (exiting O). Thus I and O define the "embedding" of the right part (C) in the host graph. Thus the way in which the "embedding transformation" is specified in the grammar is extremely simple.

An H-graph grammar is "ambiguous" if the language that it generates contains a graph with two or more distinct derivations.

Definition - Legal GILT graph

Let Q be the base grammar defined in chapter five, figure 5.5 and W be the communications grammar of figure 5.20 in the same chapter. Q is a comprehensive definition both of the control flow structures used in GILT and the connection points for graphs generated using the communication grammar "W", which is a comprehensive definition of the legal communication constructs of the language.

Let two node and arc vocabularies, V_M and V_A such that :

$$V_M = V_{MQ} \cup V_{MW},$$

$$V_A = V_{AQ} \cup V_{AW}.$$

where V_{MQ} and V_{MW} are the node vocabularies of Q and W respectively, and V_{AQ} and V_{AW} are the arc vocabularies of Q and W respectively.

A graph G_1 over (V_M, V_A) is a legal level-1 GILT graph if it can be formed by the union of two graphs B_1 and C_1 such that :

B_1 is a level-1 base H-graph, defined by grammar Q.

C'_1 is a set of n communications graphs defined by grammar W.

$$C'_1 = \{C_{1,1}, C_{1,2}, \dots, C_{1,n}\}$$

All $C_{1,i} \in C'_1$ are defined to be disjoint s.t.

for all $C_{1,j}, C_{1,k} \in C'_1$,

$$a \in C_{1,j}, a \notin C_{1,k}.$$

The "total communications graph" C_1 is formed by the union of all disjoint $C_{1,i}$'s :

$$C_1 = \bigcup_{i=0}^{n-1} C_{1,i}$$

N_{B1O} is the set of all nodes labelled "CHANNEL OUTPUT PORT" in B_1 .

N_{B1I} is the set of all nodes labelled "CHANNEL INPUT PORT" in B_1 .

Similarly,

N_{C1O} is the set of all nodes labelled "CHANNEL OUTPUT PORT" in C_1 .

N_{C1I} is the set of all nodes labelled "CHANNEL INPUT PORT" in C_1 .

so that,

N_{B1O} is a proper subset of B_1 , N_{B1O} a proper subset of B_1 ,

and,

$$N_{B1O} \subseteq B_1 \quad N_{B1O} \subseteq B_1,$$

If $N_{B1O} \equiv N_{C1O}$ and $N_{B1I} \equiv N_{C1I}$.

$G_1 = B_1 \cup C_1$ and is a legal GILT level-1 graph.

Let G_k be a graph over (V_M, V_A) . G_k is a level- k ($k \geq 1$) GILT graph if all the values of nodes in G_k which are graphs are legal level $(k-1)$ GILT graphs, and if two graphs B_k and C_k can be generated such that :

B_k is a base H-graph, defined by grammar Q .

C'_k be a set of n communications graphs defined by grammar Q .

$$C'_k = \{C_{k,1}, C_{k,2}, \dots, C_{k,n}\}$$

All $C_{k,i} \in C'_k$ are defined to be disjoint s.t.

for all $C_{k,j}, C_{k,k} \in C'_k$,

$a \in C_{k,j}, a \notin C_{k,k}$.

The total communications graph C_k is formed by the union of all disjoint $C_{k,i}$'s :

$$C_k = \bigcup_{i=0}^{n-1} C_{k,i}$$

N_{BkO} is the set of all nodes valued CHANNEL OUTPUT PORT in B_k .

N_{BkI} is the set of all nodes valued CHANNEL INPUT PORT in B_k .

Similarly,

N_{CkO} is the set of all nodes valued CHANNEL OUTPUT PORT in C_k .

N_{CkI} is the set of all nodes labelled CHANNEL INPUT PORT in C_k .

so that,

$$N_{BkO} \subseteq B_k, N_{BkI} \subseteq B_k,$$

and,

$$N_{CkO} \subseteq C_k, N_{CkI} \subseteq C_k,$$

$$N_{BkO} \equiv N_{CkO} \text{ and } N_{BkI} \equiv N_{CkI}.$$

$G_k = B_k \cup C_k$ and is a legal k -level GILT graph.

A generalised legal GILT graph is one such k -level GILT graph where k is finite.

The definition of the H-graph and grammar are based on that in (Pratt, 1971). It differs from Pratts's in that it uses a different method for the creation of graphs which cannot be generated using a context-free graph grammar. Instead of using labelling and reduction, as used by Pratt, it uses a simple set union method in which a complete graph is formed from two partial graphs (the base and communication graphs). The graphs are joined by graph union at the nodes with labels "CHANNEL INPUT PORT" and "CHANNEL OUTPUT PORT". Various restrictions ensure that such nodes are fully connected into the final graph and that no unconnected nodes exist

Appendix 3

Published Work

This appendix contains published work by the author which is relevant to the subject of the thesis. Work is included in the following order; (Roberts and Samwell, 1989), (Roberts and Samwell, 1990), (Roberts, 1990a), (Roberts 1990b).

A VISUAL PROGRAMMING SYSTEM FOR THE DEVELOPMENT OF PARALLEL SOFTWARE

M. Roberts and P. M. Samwell

The Centre for Information Engineering, City University, London, U.K.

ABSTRACT

This paper describes GILT, a visual programming language for transputer based multiprocessors, which is under development at City University. GILT programs take the form of hierarchical graphs in which nodes are processes visually represented by icons and edges are either control flow or inter-process communication paths. Users of GILT interactively build programs using a mouse at a high resolution workscreen. Only when a program has been completely specified in terms of communicating processes need the user enter conventional program text for the lowest level processes. GILT programs may then be compiled into occam for execution on transputer arrays. This approach to concurrent programming has a number of advantages, which include increased speed of interaction, a much higher information transfer bandwidth between man and machine, and a more natural representation of parallelism than is allowed by conventional textual languages.

GRAPHICAL TOOLS FOR CONCURRENT PROGRAMMING

Graphical tools for concurrent program development and analysis have the potential to make parallel programming a much easier task. Backus (1) notes that concurrency introduces an "extra dimension" to programming not present in conventional von-Neumann type sequential languages. While textual languages appear reasonably well suited to the demands of sequential programming, textual concurrent programming languages force programmers to express their thoughts of multiple interacting threads of execution in terms of one dimensional textual strings. In these textual representations certain information (specifically concurrency) is well hidden and its extraction requires considerable mental effort. It is therefore natural to use multidimensional, graphical representations of parallelism, rather than textual representations.

The increased representational power of graphical views of concurrency can be recognised by the large number of diagrams used in papers on the subject e.g. (2). It is our experience that programmers, whether implementing or analysing a program, frequently use diagrams and sketches of the multiple interacting processes in a large concurrent program. More formal design methodologies for concurrent systems, such as Mascot (3) or Harel's work on Statecharts (4), can be seen as supporting this view.

It is well known that pictures provide a much higher information transfer bandwidth between computer and man than does text, hence the greater understanding of graphically represented systems. Quoting Raeder (5) "The best way to give a programmer an idea of the parallelism obtained by design is probably to devise a graphical display of the program in action that highlights the multiple

concurrent actions". Surely a programming language that highlights multiple concurrent actions is even more desirable!

Graphical tools for concurrency fall into two main areas - program visualisation (PV) tools and visual programming (VP) tools. Program visualisation tools turn a textual or non-graphical representation of a program into a visual representation showing aspects of the program's structure or behaviour. Systems producing static or dynamic views have been developed, and it has been demonstrated that many facets of parallel program structure and behaviour are excellently represented by graphical means. Some recent parallel program visualisation systems include GRAIL (6), TREE (7), MONA (8) and (9). By contrast, visual programming systems make use of graphical structures as the program input medium. While many systems have been produced for sequentially based, object oriented, or data structure programming languages surprisingly few concerned with explicit concurrency exist. Of these few, some have used the naturally visual data flow model (10), (11). Other approaches have used visual programming for the specification of the program's interprocess communication structure and the interaction between the various component processes forming a parallel program, with the specification of the functionality of the program's component processes performed with a conventional von-Neumann language. STILE (12) and Poker (13) are typical of this approach.

MATHEMATICAL MODELS OF PARALLELISM

Mathematical models of parallelism e.g. (14), (15) allow the application of formal methods to program design and development and as such are a necessary basis for concurrent languages. They have obvious applications in the design of safety critical and/or real time systems in which program correctness is a central issue. However, few visual programming systems have incorporated such rigorous methodologies. The unification of visual programming techniques and languages based on mathematical models of parallelism is obviously desirable.

VISUAL FORMALISMS

GILT is a visual programming system based on the occam (16) computational model. GILT programs take the form of hierarchical graphs in which nodes are processes and edges represent control flow or interprocess communication. Nodes contain further, lower level, GILT graphs or simple occam code. The graph model used allows the construction of programs consisting of small, potentially provable occam processes connected together in a consistent and visual way. Such an approach is similar to that used in (17). In a typical session a user would initially sketch a design of channel connected processes, similar to the sketches produced by many occam users in program development.

The functionality of the individual processes defined would then be expressed in further, lower level, GILT graphs until the user had satisfied himself that a sufficiently small level of granularity had been reached. Simple occam code would then be written for the lowest level processes. This "top down" approach is not equivalent to pure functional design, but produces hierarchical sets of functionally related processes which are expressed in a graphical sense by the use of GILT's visual graph notation.

THE GILT SYSTEM

GILT users interactively build program graphs using a mouse at a high resolution workscreen. Graphs are visually displayed by the use of icons representing nodes and graphical 'links' showing edges. Such visual representations have numerous advantages. For example, it is well known that human minds are strongly visually oriented and that people acquire information at a much higher rate by discovering graphical relationships in complex pictures than they do by reading text (5). Instant random visual access to any part of a picture is provided by our eyes whereas text is an essentially sequential medium. Pictures also provide many more dimensions of expression than do words, and allow us to use a host of familiar visual symbols for the expression of concepts. Gilt icons do not provide functionality - for example, they are not arguments to pre-defined processes, as icons have been in some previous systems. Rather they provide a visual description of the functionality of the process that they represent. Box based visual programming systems e.g. (11), (18) do not allow the use of real world symbols and thus iconic systems with user defined symbols potentially provide much more program related information. Graphical program representations do not however replace textual descriptions for the expression of abstract concepts. Rather textual and graphical representations complement each other in that textual representations can convey information difficult to express graphically and vice versa.

Real time systems and GILT

Real time systems are inherently parallel - and so a language based on parallel communicating modules is ideal for real time applications. Equally desirable is a formal basis to such a language allowing the application of formal methods for program design and development. In GILT or occam based real time software, applications are decomposed into many small communicating processes, each dealing with one aspect of the system's real time behaviour. The formal aspects of occam should allow the behaviour of such small processes to be reasoned about. However, textual languages obscure parallelism and communications between parallel program modules, making the writing of such parallel programs an unnecessarily difficult task. The GILT approach is intended to give the programmer the advantages that occam offers in terms of modularity and formality, while adding features intended to make the coding of communications structure and parallelism an easier task.

GILT Graph Components

Gilt Graphs are composed of a small number of basic components :

Process Icons represent process nodes analogous to occam processes (See Figure 1). Each process node contains either a few lines of occam code or an internal GILT graph. The appearance of process icons may be altered by the programmer to give an indication of the functionality of the process node that the icon represents. Figure 2 shows the internal GILT graph of the process node represented by the icon in Figure 1.

Control flow links and as many ports as are required for communication with other process nodes are arranged around the outside of the icon. Channel links connect ports to ports or ports to Channel stub icons (see Figure 2). Ports are representations at a higher level of abstraction of the channel stub icons present at a lower level of abstraction. Thus, the display process of Figure 3 has one input port, which corresponds to the single channel stub icon shown in Figure 2. Communication between levels of abstraction is implemented in this manner. Note that, as in occam, the connection of two ports by a channel link does not indicate that the processes owning the ports will communicate, only that they may.

Control flow stubs indicate the connection of control flow to the level of abstraction above the current level in the hierarchy of graphs. Only one control flow input and one control flow output are allowed at any level of abstraction. The appearance of control flow links, ports, channel links and channel stub icons is at present fixed, and is not alterable by the programmer. Figure 2 illustrates all the basic components of a GILT graph.

Figure 3 shows a GILT display representing a pipeline structure for the calculation of mandelbrot sets. Five concurrent processes are shown - a display process, a controller process, two worker processes (which are identical) and a special end pipeline process. The display process handles the display of information passed to it by the controller process, which also allocates work to and collects work from the worker processes. Worker processes perform work sent to them by the controller process, as well as routing work to and from other worker processes in the pipeline. The end pipeline process also performs work, but does not need to do any through routing of work for other processes. Concurrency in the pipeline is shown by the control flow forking to pass through each of the processes shown in parallel. In sequential control flow, the control flow links would connect processes serially.

Each node or edge in the graph may also have a textual comment associated with it. This allows the pictorial representation of a program to have text associated with it in the same way that we might annotate a diagram. Allowing free-hand drawing of arrows and other graphical annotations may be desirable in future, as such sketches are frequently used in both program development and documentation.

GILT graph structure rules

It is not enough that the graph components simply be defined - some restrictions must be placed on the classes of graph that may be constructed, as certain concepts that may be visually expressed can have no meaningful implementation in occam. These restrictions are enforced by the use of structure directed editing. Structure directed editing is equivalent to the syntax directed editing methods employed with conventional languages, but instead of acting on the syntactic rules

associated with a textual language, it uses structure rules associated with the graph model. Our set of structure rules is certainly not complete in that it is possible to produce erroneous GILT programs. An example of an error would be a deadlocking program. It is difficult to devise restrictions to the graph formation rules which preclude such errors without imposing unnecessary limitations on valid programs. However, logical analysis may be useful in "highlighting" possible problem areas for programmer consideration.

Our set of graph structure rules are at present represented in a verbal and non-formal way. A good example of one such rule would be our 'non-recursive' rule - "No node may have itself as a descendant". This rule is a simple reflection of implementation restrictions - occam does not presently support recursion. Other rules disallow the production of classes of incomprehensible program structures, for example - separate threads of execution are not allowed to pass through the same instance of a process. (We define a thread of execution as a sequence of instances of execution of processes. The number of processes in the thread must be finite and non-zero.) We are currently investigating the formal representation of such rules. Harel (4) has discussed formal descriptions of visual representations and it is easy to see the relationship between Harel's Higraphs and GILT graphs. The use of a description such as that of Higraph for the GILT visual model may enable a full formal representation for the system to be developed. Certain parallels also exist between Higraph descriptions and the CSP model (14).

User Interaction with the system

Users interact with the GILT editor by means of a mouse and menu based system. Icons representing processes and i/o stubs are drawn in a special icon editing area, then dragged to an appropriate position on the screen. Icons may be "entered" to reveal detail within them. Entering an icon is equivalent to going down one level in the hierarchy to a lower level of abstraction. Text for the lowest level icons is entered from the keyboard into pop-up windows. Note that no capability to "open" icons is provided due to limited screen resolution and layout problems. Indeed, allowing the opening of nodes in hierarchical systems may be undesirable, as it encourages a programmer not to structure his program in a top-down manner. Support for concurrent editing at different levels of abstraction will be supported in later versions of the system editor, although the current editor supports only editing at one (variable) level. Control flow and channel links are also defined with the mouse. Structure directed editing prevents connection of channel stubs to control flow, placement of icons on top of one another and other similar problems. Figure 3 shows a GILT display.

Compilation of GILT graphs

Compilation of GILT graphs is a two stage process. Firstly, GILT graphs are compiled into occam code. This code is then compiled via a standard occam compiler to produce transputer executable code. A parse tree is produced from the GILT program graph, allowing easy application of program transformation techniques. Initial versions of the compiler produce code for a single

processor only. Code for multiple processors will be produced by the use of a visual process-to-processor mapping tool in which processes are grouped together and interactively assigned to processors. A similar graphical tool, Gecko, has been produced for transputer based systems (19). However, GILT's hierarchical graph model allows a novel approach to the mapping problem. Processes at particular levels of abstraction are assigned to groups of processors. The assignment of processes to individual group members is then carried out at a lower level of abstraction. Providing group members are close to each other in terms of communication, and processes at a given level of abstraction are also close in terms of communications, then an efficient mapping should be achieved.

ANIMATION

Although animation is a powerful aid to the comprehension of concurrency, few previous systems have made use of animation to represent explicit concurrency. Systems that do so include (9), PIE (20), MONA (8), Tree and Graph (7). (9) uses animation to display various facets of parallel program behaviour for debugging purposes. PIE provides a graphical animated view of objects and their relationships within a parallel environment and MONA displays a process event trace graphically, updating this information to provide a "movie" of the actions taken by a distributed system. Tree and Graph provide animated program and machine level views of dataflow programs in execution. In addition to these systems, a number of papers on visual programming have proposed the use of animation for the display of concurrency (5), (21), (22).

GILT is already a visual programming language and so there is no need to create a different view of the program as is the case with program visualisation systems. We only need to select appropriate views of the program at particular levels of abstraction and overlay information on top of the existing views. For example, concurrently executing processes may be shown by highlighting and data may be shown moving down channels by tokens. Synchronised communication may be shown by first highlighting stubs that are ready to communicate. Only when two connected stubs are highlighted can a token move graphically down a channel to indicate data transmission between the processes. Use can also be made of a multi-windowing environment to provide views of the program in execution at different levels of abstraction.

The extraction of information from multitransputer systems is interesting, but beyond the scope of this paper. We do however, plan to extract information from running transputer systems. West (23) has investigated the use of program transformation for this purpose and Tang (7) discusses a variety of methods that could be used to obtain information for use in animated program views.

GRAPHICAL PROGRAM TRANSFORMATION

As occam is a language based on algebraic laws, it is possible to transform one form of an occam program into another. This technique is known as program transformation. Transformations may for example be applied to programs to improve their efficiency. However, with a textual representation it is difficult to obtain an overview of the

transformed program and the net effect of the transformation. A graphical program model provides a framework for transformation in which the results of transformations can easily be seen. As GILT graphs allow a freer representation of parallelism than is obtainable with textual programming languages, it is hoped that programmers may feel encouraged to encode their programs in a highly parallel (ideally maximally parallel) manner. When the program is to be scheduled for maximum real time efficiency over a number of processors, a maximally parallel version of the program is not necessarily desirable - as the processors will have to perform large amounts of context switching. A technique suggested by Roscoe and Hoare (24) might be to take a maximally parallel version of the program and use the symmetry and associative laws of the occam PAR constructor to divide the task into groups of processes suitable for running on single processors in a given network. Some of the parallelism within these groups could then be eliminated. Such an approach is ideal for graphical implementation with users being able to directly view the effect of the restructuring on their program. Other program transformation approaches could be equally informative when visually represented.

SUMMARY

This paper has described GILT, a concurrent visual programming system for transputers. GILT programs are represented as visually displayed hierarchical graphs, a notation which has many advantages for parallel programming and provides a basis for program transformation, animation, and other desirable techniques. The system is currently being implemented on a Sun 4/110 workstation. The user interface and graph editing systems are completed and a compiler for GILT graphs is under development. Ongoing work centres on four main areas. Firstly, the development of an efficient compiler for GILT graphs. Secondly, the extraction of debugging information from transputer systems and its overlay on GILT graphs. Thirdly the practicality and uses of visual program transformation, and finally on a formal description of the GILT graph model.

REFERENCES

1. Backus, J., 1978, "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," Communications of the ACM, vol. 21, No. 8, August 1978.
2. Samwell, P.M., 1986, "Experiences with occam for simulating systolic and wavefront arrays," Software Engineering Journal, Vol. 1, No. 5, September 1986.
3. Simpson, H., 1986, "The Mascot Method," Software Engineering Journal, Vol. 1, No. 3, May 1986.
4. Harel, D., 1988, "On Visual Formalisms" Communications of the ACM, Vol. 31, No. 5, pp. 514-530, May, 1988.
5. Raeder, G., 1985, "A Survey of Current Graphical Programming Techniques," IEEE Computer, vol. 18, No. 8, pp. 11-25, August 1985.
6. Stepney, S., 1987, "GRAIL : Graphical Representation of Activity, Interconnection and Loading," Proceedings 7th Occam User Group International Workshop on Parallel Programming of Transputer Based Machines, Grenoble, France, September 14-16, 1987. T. Muntean (Ed.)
7. Tang, H. W. H., 1987, "Monitoring Tools for Parallel Systems", Technical Report No. UMCS-87-12-3, Department of Computer Science, University of Manchester, Manchester M13 9PL, England, 1987.
8. Joyce, J., Lomow, G., Slind, K. and Unger B., 1987, "Monitoring Distributed Systems," ACM Transactions on Computer Systems, Vol. 5, No. 2, May 1987.
9. Zimmermann, M., Perrenoud, F., and Schiper, A., 1988, "Understanding Concurrent Programming Through Program Animation" ACM SIGCSE Bulletin, Vol. 20, Part 1, November 1988.
10. Kerner, H. and H. Rainel, 1986, "EDDA : A Language based on Petrinets and the Dataflow Principle for the development of Parallel Programs," Microprocessing and Microprogramming, Vol 18 (Proceedings of the 12th Annual Euromicro Symposium), pp. 299-306, 1986.
11. Cox, P. T. and I. J. Mulligan, 1985, "Compiling the Graphical Functional Language PROGRAPH," Proceedings of the 1985 ACM SIGSMALL Symposium on Small Systems, pp. 34-41, May 1-3, 1985.
12. Taneja, S. and B. W. Weide, 1986, "Graphical Description and Run-Time Environments for Real-Time Software," Proceedings of the ACM 14th Annual Computer Science Conference CSC '86, vol. 4-6, pp. 205-211, Ohio, USA, 1986.
13. Snyder, L., 1984, "Parallel Programming and the Poker Programming Environment," IEEE Computer, vol. 17, No. 7, pp. 27-36, July 1984.
14. Hoare, C. A. R., 1985, "Communicating Sequential Processes," Prentice Hall International Series in Computer Science, 1985.
15. Milner R., "A Calculus of Communicating Systems," Lecture Notes in Computer Science 92, Springer-Verlag, New York, 1980.
16. May, D., 1987, "Occam 2 Language definition," Inmos Ltd., 1000 Aztec West, Almondsbury, Bristol BS12 4SQ, U.K., February 13, 1987.
17. Yau, S. S. and P. C. Grabow, 1981, "A Model for Representing Programs using Hierarchical Graphs," IEEE Transactions on Software Engineering, Vol. SE-12, No. 6, November 1981.
18. Pong, M. C. and N. Ng, 1983, "PIGS - A System for Programming with Interactive Graphical Support," Software Practice and Experience, vol. 13, pp. 847-855, 1983.
19. Stephenson, M. and Boudillet, O., 1988, "Gecko: A Graphical tool for the modelling and manipulation of occam software and transputer hardware topologies," Occam and the Transputer - Research and Applications (Proceedings of the 9th occam user group technical meeting 19-21 September 1988, Southampton, U.K.), C. Askew (ed.), IOS 1988.
20. Segall, Z. and L. Rudolph, 1986, "PIE : A Programming and Instrumentation Environment for Parallel Processing," IEEE Software, vol. 18, No. 11, November 1985.
21. Rovner, P. D. and D. A. Henderson Jr., 1969, "On the Implementation of AMBIT/G : A Graphical Programming Language", Proceedings of the International joint conference on Artificial Intelligence, pp. 9-19, Washington D.C., USA, May 7-9, 1969.
22. Myers, B. A., 1988, "The State of the Art in Visual Programming and Program Visualisation," CMU Report No. CMU-CS-88-114, Carnegie Mellon University, Pittsburgh, PA, USA, February 1988.
23. West, A. J., "Monitoring Distributed Occam Programs," Thesis for the degree of Doctor of Philosophy in the Faculty of Science, University of Manchester Dept of Computer Science, Manchester, U.K., 1987.
24. Roscoe, A. W. and Hoare, C. A. R., "The Laws of Occam Programming," Technical Monogram PRG-53, Oxford University Computing Laboratory Programming Research Group, 8-11 Keble Road, Oxford, England.

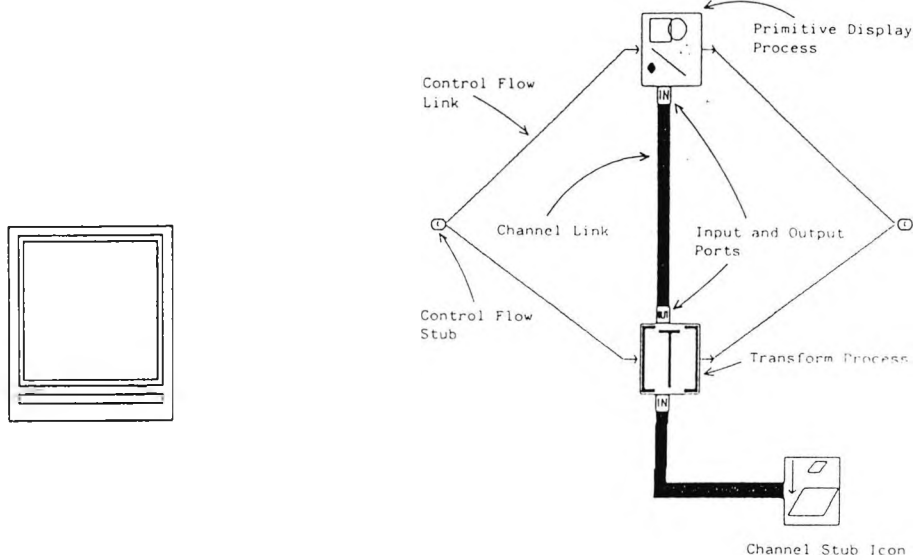


Figure 1 - A process icon representing a display managing process

Figure 2 - The internal detail of the process represented by the icon of Figure 1. Two parallel processes, a co-ordinate transforming process and a primitive display process, are shown.

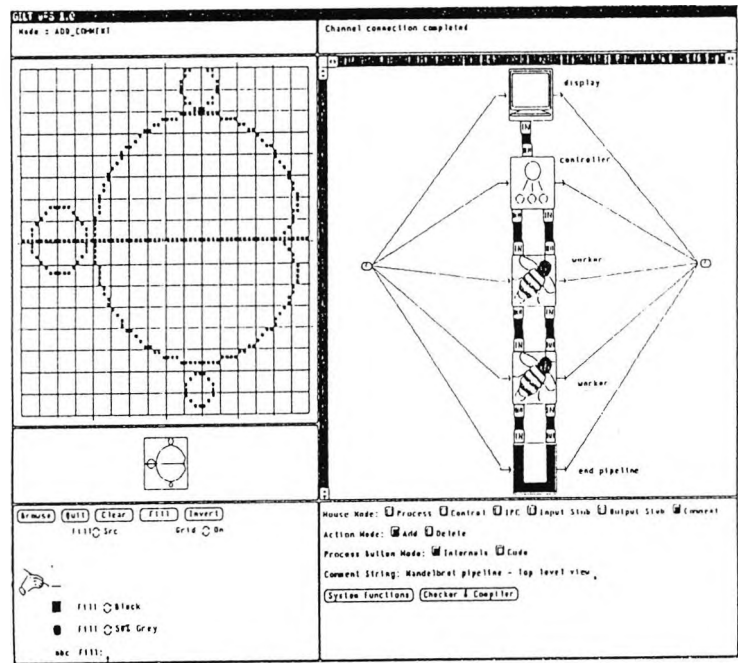


Figure 3 - GILT system display for the Mandelbrot program. On the left hand side of the display is the icon editing area. On the right hand side is the program itself. Five communicating concurrent processes are shown.

A Visual Programming System for the Transputer

M. Roberts and P.M. Samwell,

The Centre for Information Engineering,

City University,

Northampton Square,

London,

EC1R OAE,

U.K.

Email M.ROBERTS@uk.ac.city

Abstract

The difficulty of producing reliable safety-critical software is widely appreciated. The major problems arise from the complexity of realistic systems and their inherent requirements of interaction with concurrent, real-time and possible non-deterministic aspects of their environment. The software engineer's best strategy for maximising confidence in solutions is the use of tools which have a rigorous mathematical background. Recent visual programming techniques can also aid the software engineer by facilitating the management of complexity in concurrent software development and by providing a better human interface to the programming process. However, few visual programming systems have incorporated mathematical models of parallelism. They have instead focused on a less formal and more humanistic approach. The unification of visual programming techniques and parallel languages based on mathematical models is obviously desirable.

This paper describes the visual programming language GILT (Graphical Language for Transputers) and discusses how the application of GILT-like languages to the development of parallel software can further the development of explicitly parallel programming and widen the acceptance of the transputer as a basic component for the production of parallel systems. Compiling GILT visual programs into occam as well as new developments in the GILT visual graph model are discussed. Applications in both computer science education and concurrent program development are given.

On the use and facility of graphics in concurrent programming

It is well known that visual, as opposed to textual, representation is a powerful tool for comprehension and expression of complex designs - in particular many aspects of parallel systems. We have found that programmers of transputer based systems often draw diagrams of processes and communications channels as well as the 'rack diagram' commonly found at transputer sites. More formal design methodologies, such as Mascot [Simpson86] or Harel's work on Statecharts [Harel88], can be seen as further evidence as to the facility of graphics in a concurrent environment.

Visual (graphical) programming languages not only achieve a very high information transfer bandwidth between human and machine, but they also give a far more natural description of parallelism than is achieved by conventional textual languages. This more natural description of parallelism aids the software development process by increasing use of the left hemisphere of the brain, which is currently under-used in programming [Shu88]. It has been noted that graphical program development tools are particularly efficient in computer science education. A good application for graphical parallel program development tools lies in the initiation of the conventional sequential programming community in concurrent programming techniques. The tools can help to widen acceptance of parallel processing by improving the human computer interface in concurrent programming practice.

In addition, graphical tools may be able to provide a highly humanistic interface to formal mathematical models of parallelism (e.g. [Hoare85, Milner87]), which allow the application of formal methods to program design and development and as such are a necessary basis for concurrent languages. However, most visual programming systems have yet to incorporate such rigorous methodologies. The unification of visual programming techniques and languages based on mathematical models of parallelism is obviously desirable.

These reasons, amongst others, have lead us to believe that the visual expression of concurrent programming structures is a realistic tool for code development for concurrent systems, particularly for those systems based on the naturally visual communicating processes model.

Recent and related work

Recent work on graphical program development tools has centred in two main areas - program visualisation and visual programming. Program visualisation tools turn a textual or non-graphical representation of a program into a visual representation showing aspects of the program's structure or behaviour. By contrast, visual programming systems make use of graphical structures as the program input medium. While many systems have been produced for sequentially based, object oriented, or data structure programming languages surprisingly few concerned with explicit concurrency exist. Some relevant concurrent visual programming and program visualisation systems are discussed in [Roberts89]. Two recent and interesting systems are described in [Mourlin89] and [Crowe89]. Such tools have demonstrated that both program visualisation and visual programming have the potential to make parallel programming an easier task.

Visual programming with GILT

GILT is a visual programming system based on the occam [May87] computational model. Programs are built at a workstation by interactive construction of hierarchical graphs in which nodes are processes and communication facilities, with edges showing flow of control and inter-process communication. Nodes are visually represented by icons with textual labels, while edges are shown using a variety of different line styles dependent on the edge's function. Processes may have graphical sub-processes or, at the lowest levels of abstraction, may be directly expressed in occam. The graph model used allows the construction of programs consisting of small, potentially provable occam processes connected together in a consistent and visual way. A user may initially sketch a design of channel connected processes similar to the sketches produced by many occam users during program development. The functionality of the individual processes may then be expressed in further, lower level, GILT graphs until the user is satisfied that a sufficiently small level of granularity has been reached. Simple occam code can then be written for the lowest level processes. This "top down" approach is not equivalent to pure functional design, but produces hierarchical sets of functionally related processes which are expressed in a graphical sense by the use of a visual graph notation. Alternatively, programs can be constructed by the "snapping together" of standard program components stored in a process library. This approach is similar to the conventional engineering practice of bottom up design and has advantages for code reuse. Realistically, both approaches may be used in practical program development. Previous CSP based visual programming systems (such as [Pong86]) have not included control flow within their graph model. The inclusion of control flow into the graph model allows visual programming to proceed to a lower level of abstraction than is possible with the use of pure communicating processes visualisation models. In certain situations, however, viewing control flow may be confusing. GILT therefore allows users to hide control flow from sight, so that graphs may be viewed and constructed using a pure communicating processes model. This facility is particularly useful when all the processes at a particular level of abstraction are in parallel, for example in a systolic array.

GILT's icons do not provide functionality - they are not, for example, arguments to pre-defined processes as icons have been in some previous systems. Rather they provide a visual description of the functionality of the node which they represent. The pictographic representations used in GILT do not, however, replace textual descriptions for the complete representation of abstract concepts - GILT is based on a mixed textual and graphical model with text and graphics complementing each other. Textual process names, comments and variable definitions (just some examples) are as important in a program as is the program's visual information. GILT's close integration of control flow, inter-process communication and text within a single unified graph model addresses many of the problems associated with both visual programming and concurrency. As illustration, GILT's text-graphic integration yields a convenient graphical representation for guarded execution of processes lacking in previous concurrent visual programming systems. (Figure 1)

Users interact with the GILT editor by means of a mouse and menu based system. Icons representing processes and i/o stubs are drawn in a special icon editing area, then dragged to an appropriate position on the screen. Icons may be "entered" to reveal detail within them. Entering an icon is equivalent to going down one level in the hierarchy to a lower level of abstraction. Text for the lowest level icons is entered from the keyboard into pop-up windows. Note that no capability to "open" icons is provided due to limited screen resolution and layout problems. Indeed, allowing the opening of nodes in hierarchical systems may be

undesirable, as it encourages a programmer not to structure programs in a top-down manner. Support for concurrent editing at different levels of abstraction will be supported in later versions of the system editor, although the current editor supports only editing at one (variable) level. Control flow and channel links are also defined with the mouse.

GILT graph components

GILT graphs are composed of a small number of basic components - Process Icons, Guard Icons, Comments, Variable Declaration Icons, Passed Variable Icons, Control Flow Links, Channel Stub Icons, Channel Links and Control Flow Stub Icons. Readers are directed to [Roberts89] for a fuller description of the most basic components. Since then, a number of refinements to the graph model have introduced several new components to the above list :- Guard Icons, Variable Declaration Icons, and Passed Variable Icons. We plan to add further refinements to the model, such as control flow switches corresponding to the occam 'if' and 'case' statements. GILT's icons have also been amended to include a textual label, similar to the textual icon labels used in Hi- Visual [Yoshimoto86]. This system allows users to easily determine the function of iconic symbols new to them, while still allowing the use of the icons for quick information recognition. Figure 2 shows current symbols within the graph model. Figure 3 shows a trivial graph, and its equivalent in occam.

Ensuring graph correctness

It is possible to express concepts visually that can have no meaningful implementation in occam. One such construct would be the connection of control flow to a Channel Stub Icon. Restrictions to the structures that may be evolved are enforced by the use of structure directed editing. Structure directed editing is equivalent to the syntax directed editing methods employed with conventional languages, but instead of acting on the syntactic rules associated with a textual language, it uses structure rules associated with the graph model. The set of structure rules that may be evolved are enough to produce syntactically correct programs. It is difficult to devise restrictions to the graph formation rules which preclude logical errors without imposing unnecessary limitations on valid programs. However, the development of compiler for GILT graphs based on the graph reduction principle has highlighted a very plausible approach to the checking of graphs, which is even "smart" enough to enable suggestions be made as to possible correct program structures within an erroneous framework.

Harel [Harel88] has discussed formal descriptions of visual representations and it is easy to see the relationship between Harel's Higraphs and GILT graphs. Although it is possible to represent GILT graphs using Harel's Hi-Graph theory, we are using a context free graph grammar for the formalisation of GILT graphs.

Compilation of GILT graphs

Compilation of GILT graphs is a two stage process. Firstly, GILT graphs are compiled into occam code. This code is then compiled via a standard occam compiler to produce transputer executable code.

A new GILT compiler which works on a reduction principle relying on the laws of occam [Roscoe86], particularly the symmetry and associativity of occam constructor processes, is being written.

A number of graph reductions have been developed which may be applied to GILT graphs. These reductions replace existing graph structures with a new node in the graph whose attribute describes the original structure. For example, a parallel reduction rule specifies that two nodes wired in parallel may be reduced to a single node with an attribute list representing the two original nodes executing in parallel. Due to the associativity of the PAR constructor, it is possible to reduce any n-branch parallel structure into a single node with a textual attribute equivalent to the parallel structure.

Equivalents for the reduction of SEQ constructs exist. Reduction of ALT constructs requires several simple reductions. The compiler may be modified to recognise different visual structures by the addition of new reductions or by modification of existing ones. Communication between process nodes is easily compiled by replacing instances of processes in the textual attribute list of a reduced node with a channel definition and process instances with the generated channel name as a passed parameter.

The reductions are encoded as a set of productions. These productions of a grammar define the syntax of the graphs. Any fully reducible graph is therefore syntactically correct. The productions are encoded in Prolog, allowing them to be repeatedly applied to a Prolog rule base containing a description of the graphs. The Prolog rule base description is produced by the graph editor from its internal database describing graphs, with the editor performing tokenisation usually associated with the initial pass of a compiler over a conventional textual language. Erroneous graphs will not reduce fully; they are left partly reduced, with the erroneous structure remaining in its original form. Error detection is therefore greatly simplified, and is just a matter of checking to see if graphs have been reduced at every level of abstraction. Propagation of errors back into the visual representation of the graph is also rather easy - outstanding links and nodes can be related back to the original graph and, for example, highlighted.

Of some concern is the slowness of the compilation process due to the Prolog matching process. It may be advantageous to re-code the reduction process in C once it has been thoroughly explored.

Productions may also be applied to graphs for the purpose of program transformation. When program transformation is applied to a textual language it is difficult to obtain an overview of the net effect of the transformation. However, with our graphical programming model, simple transformations may be applied to the rule base and the results displayed graphically. This approach provides a good interface to the transformation process.

Initial versions of the new compiler will produce code for one processor only, though the use of a visual language allows easy use of a visual process to processor mapping tool similar to Gecko [Stephenson88]. It is also possible to approach the mapping problem in a topdown graphical manner, assigning groups of processes to 'super groups' of processors, then carrying out stepwise refinement. Performance feedback in the form of coloured processor and channel activity coding like that used in Grail [Stepney87] can be used to guide the assignment process.

Graphical debugging and performance analysis

The views that GILT provides of parallel programs are static ones. It is easy to see that animated execution of explicitly concurrent programs has advantages for debugging and performance analysis. As a view of the program is already in existence any future graphical debugging or performance analysis system does not need to generate its own visualisation. Information can simply be overlaid on top of the existing view. This is not the case with program visualisation systems, in which computation of the view often forms the largest part of the system. Animation of important concurrent concepts such as synchronised communication may help to reveal deadlocks and other related bugs. A variety of methods can be used to obtain suitable information for such an animation. The issues involved in the extraction of suitable information are more fully discussed in [West87], while Zimmermann [Zimmermann88] discusses the issues involved in animation.

Summary

This paper has described GILT, a concurrent visual programming system for transputers. GILT provides an abstract, hierarchical, visual program development system with close textual and graphical integration. GILT relies upon occam's mathematically based syntax and semantics. We aim to provide designers of parallel software systems with a more tractable and, therefore, a more productive design environment without sacrificing the rigour of the underlying level of implementation.

The system is under development on a Sun 4/110 workstation. The user interface and graph editing systems have been completed and a new compiler for GILT graphs, based on the reduction principle, is under development. The user interface and graph editing systems are written in C, whilst the compiler is currently written in Prolog. Our primary priority at present is the completion of the new compiler and the back propagation of errors into the editing system.

References

- [Crowe89] Crowe, W.D., Hasson, R. and Strain-Clarke, P.E.D., "A CASE tool for designing deadlock free occam programs", *Developing transputer applications*, J. Wexler (ed.), OUG-11, September 1989.
- [Hoare85] Hoare, C. A. R., 1985, "Communicating sequential processes", *Prentice Hall International Series in Computer Science*, 1985.
- [May87] May, D., 1987, "Occam 2 language definition", *Inmos Ltd.*, 1000 Aztec West, Almondsbury, Bristol BS12 4SQ, U.K., February 13, 1987.
- [Milner87] Milner R., "A calculus of communicating systems", *Lecture Notes in Computer Science* 92, Springer-Verlag, New York, 1980.
- [Mourlin89] "Graphical environment for occam programming", *occam user group newsletter*, No. 11, July 1989.
- [Pong86] Pong, M.C., "A graphical language for concurrent programming," *Proceedings of the 1986 IEEE computer society workshop on Visual Languages*, pp.26-33, Dallas, Texas, USA., June 25-27, 1986.
- [Roberts89] Roberts, M. and Samwell, P.M., "A visual programming system for the development of parallel software", *Proceedings of the Second International Conference on Software Engineering for Real Time Systems (IEE)*, pp.75-79, Cirencester, U.K., 19-20 September, 1989.
- [Roscoe86] Roscoe, A.W., and Hoare, C.A.R., "The laws of occam programming", *Technical Monogram PRG-53*, Oxford University Computing Laboratory Programming Research Group, 8-11 Keeble Road, Oxford, U.K., 1986.
- [Shu88] Nan C. Shu, "Visual programming", Van-Nostrand Reinhold, 1988.
- [Simpson86] Simpson, H., 1986, "The mascot method", *Software Engineering Journal*, Vol. 1, No. 3, pp. 103-120, May 1986.
- [Harel88] Harel, D., 1988, "On visual formalisms", *Communications of the ACM*, Vol. 31, No.5, pp. 514-530, May, 1988.
- [Stephenson88] Stephenson, M. and Boudillet, O., 1988, "Gecko: A graphical tool for the modelling and manipulation of occam software and transputer hardware topologies", *Occam and the Transputer - Research and Applications (OUG-9)*, C. Askew (ed.), pp.139-144, IOS 1988.
- [Stepney87] Stepney, S., 1987, "GRAIL : graphical representation of activity, interconnection and loading", *Parallel Programming of Transputer Based Machines (OUG-7)*, T. Muntean (ed.), IOS, September 1987.

Published work

[West87] West, A. J., "Monitoring distributed occam programs", *Thesis for the degree of Doctor of Philosophy in the Faculty of Science*, University of Manchester Dept of Computer Science, Manchester, U.K., 1987.

[Yau81] Yau, S. S. and P. C. Grabow, 1981, "A model for representing programs using hierarchical graphs", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 6, pp. 556-573, November 1981.

[Zimmermann88] Zimmermann, M., Perrenoud, F., and Schiper, A., 1988, "Understanding concurrent programming through program animation", *ACM SIGCHE Bulletin*, Vol. 20, No. 1, pp. 27-31, November 1988.

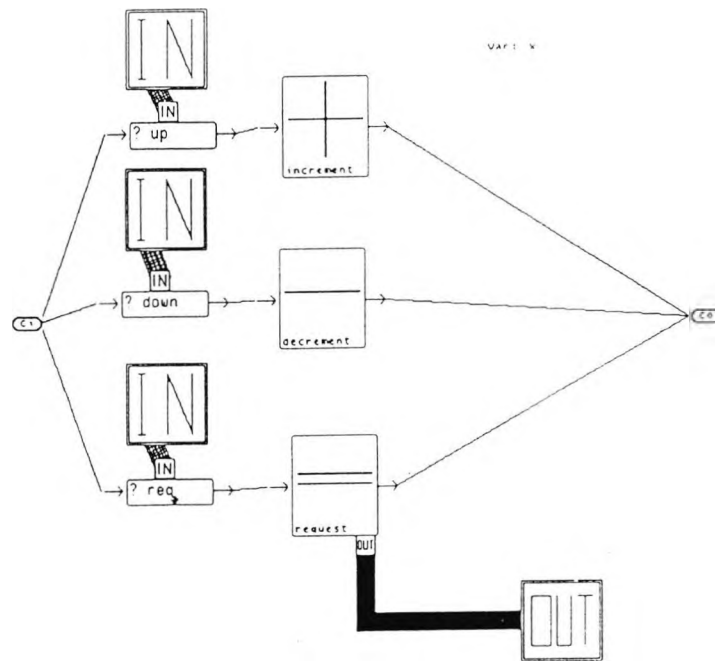


Figure 1- A simple graph showing guarded execution of three processes. Once a guard has fired, one of the processes 'increment', 'decrement' or 'request' begins operation. Consult Figure 2 for a key to the icons.

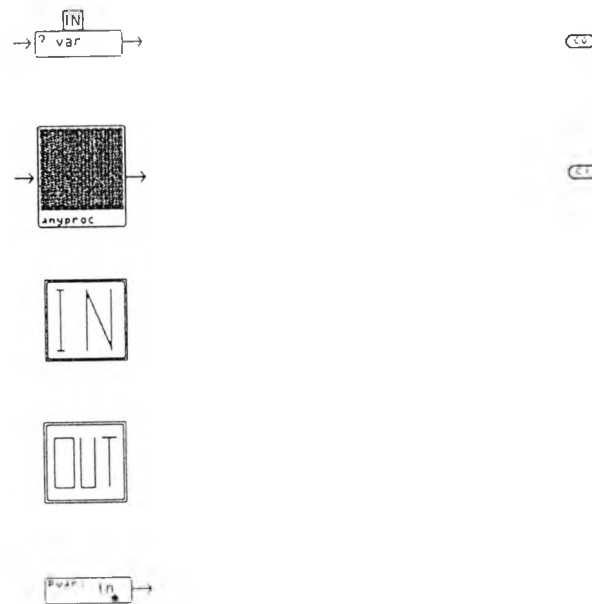
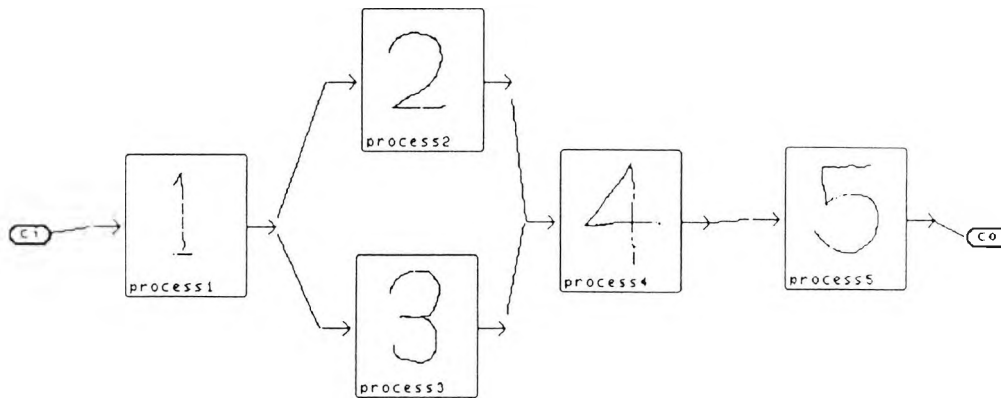


Figure 2 - Key to icons. On the left, from top to bottom - Guard Icon, Process icon, Channel Input Stub, Channel Output Stub, Passed Variable Icon. On the right, from top to bottom, Control Flow out Stub Icon, Control Flow in Stub Icon. Control Flow Links are shown as thin lines. Channel Links are shown as thicker lines.



SEQ

process1

PAR

process2

process3

process4

process5

Figure 3 - A trivial graph and its occam equivalent. The graph is purely control flow and no communication between the component processes is shown.

GRAPHICAL PROGRAM DEVELOPMENT TOOLS

a new special interest group

Mike Roberts, City University, London

At the last occam user group meeting I proposed the formation of a 'Graphical program development tools' special interest group. As the suggestion did not meet with quite the hilarity I anticipated (six people were genuinely interested) I decided to test the water with the following short piece. Anyone interested in the formation of the SIG may contact me at the address below.

What is a graphical program development tool?

Graphical program development tools do, as their name implies, use graphics in the program development process. Many experimental tools have been produced for use in all stages of the sequential software life cycle ranging from high level project management systems to low level tools using graphics in the programming process.

As yet however, few have been produced for parallel systems though many feel that graphical tools may help in that 'Holy grail' of parallel processing - the export of parallel systems and languages into the so called real world.

They fall naturally into two main areas - program visualisation tools and visual programming tools. Program visualisation is the use of computer graphics to enhance program presentation and facilitate the visualisation, understanding and effective use of programs by humans. Visual programming on the other hand is a collection of related techniques through which algorithms are expressed using various graphical representations. In short programming visualisation shows aspects of the program graphically, where as visual programming makes use of graphics as the program input medium. For initial informed introductions to both areas see references [1, 2].

But can such methods aid concurrent programming? I think that they can. Most of the reasons behind the adoption of graphics based programming tools centre on increasing the use of the left side of our brains, little used in the programming process at present. With the increased software complexity often shown in concurrent programs, it makes sense to bring as much as is possible of our underutilised brains to bear upon the task. Several recent reports from within the occam community [3, 4, 5, 6, 7] demonstrate the viability of such tools and can be seen as supporting this opinion.

If sufficient interest is expressed by members of the OUG. I will organize an initial SIG meeting at the Exeter technical meeting.

References

- [1] B. A. Myers, *The state of the art in visual programming and program visualisation*, Report Nº CMU-CS-88-144, Computer Science Department, Carnegie Mellon University, Pittsburg; presented at the British Computer Society Displays Group's Symposium on *Visual programming and program visualisation*, London, 16 March 1988.
- [2] Nan C. Shu, *Visual programming*, Van Nostrand Reinhold, New York, ISBN 0-442-28014-9, 1988.

- [3] W. D. Crowe, R. Hasson, P. E. D. Strain-Clark, *A CASE tool for designing deadlock free occam programs*, in the Proceedings of the 11th occam user group technical meeting, ed. John Wexler, *Developing transputer applications*, OUG-11, Edinburgh, IOS, September 1989.
- [4] F. Mourlin, *Graphical environment for occam programming*, occam user group newsletter N° 11, July 1989.
- [5] M. Roberts, P. M. Samwell, *A visual programming system for the development of parallel software*, in the Proceedings of the Second International Conference on Software Engineering for Real Time Systems, Cirencester, IEE, September 1989.
- [6] M. Stephenson, O. Boudillet, *GECKO: a graphical tool for the modelling and manipulation of occam software and transputer hardware topologies* in the Proceedings of the 9th occam user group technical meeting, ed. Charlie Askew, *occam and the transputer - research and applications*, OUG-9, Southampton, IOS, September 1988.
- [7] S. Stepney, *GRAIL: graphical representation of activity, interconnection and loading*, in the Proceedings of the 7th occam user group technical meeting, ed. Traian Muntean, *Parallel programming of transputer based machines*, OUG-7, Southampton, IOS, September 1987.

Mike Roberts
The Centre for Information Engineering
City University
Northampton Square
London EC1V 0HB
United Kingdom

m.roberts@uk.ac.city

GRAPHICAL PROGRAM DEVELOPMENT TOOLS SIG

Mike Roberts, City University, London

The first meeting of the Graphical Program Development Tools Specialist Interest Group meeting was held at the last OUG technical meeting. About 30 people attended the evening meeting. Notably, a strong contingent from industry attended, perhaps indicating that market forces may be moving in the direction of graphical tools.

A lively discussion with an almost 'evangelical' feel ensued, with members of the community standing up to tell the meeting about graphical tools in development, proposals for new tools, and summaries of existing tools.

Mechanical Intelligence told us briefly about the graphical configuration facilities of Express. Express seems to be one of the few tools with graphic configuration facilities in widespread use.

The benefits of Susan Stepney's GRAIL graphical performance monitoring tool were also extensively discussed, with many people feeling that GRAIL required further development as a product for the entire transputer community (not just those with T-racks). Undoubtedly such an effort would be well repaid. Interest in just such a tool was expressed by Jonathan Gulley from Thorn-EMI who instigated an interesting discussion by canvassing attendees on their views as to what a graphical program visualisation/performance monitoring tool should consist of. Many views were put forward, but a consensus was reached on the issue of portability - any such tools should be portable over a wide range of transputer based architectures and host systems.

N. Winterbottom (IBM) told the meeting of his experience with visualisation tools based on simple calls inserted into code calling display routines. Such tools are apparently not only easy to implement and use but also give a good return on the effort taken to implement them.

Other tools discussed included City's visual programming tool GILT, the Gecko configuration tool from PCL and the work at Manchester on the automatic generation of occam PROC headers from a process diagram drawing package.

Proposals for discussion at future meetings included a one day workshop on graphical tools, a more formal basis to the group, and possible demonstrations of tools at future meetings.

Anyone attending the meeting has been 'conscripted' into a list of SIG members - anyone else wishing to voluntarily 'join up' should contact me (preferably by email).

In conclusion then, a successful first meeting of the SIG with a bright future in view.

Mike Roberts	+44 71 253 4399 x3889/3877
The Centre for Information Engineering	m.roberts@uk.ac.city
City University	
Northampton Square	
London EC1V 0HB	
United Kingdom	

References and Bibliography

AE, T. and AIBARA, R. (1987).

Rapid prototyping of real-time software using Petri-nets. In: THE 1987 IEEE WORKSHOP ON VISUAL LANGUAGES. Linköping, Sweden, 1987. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 234-241.

AE, T., YAMASHITA, M., CUNHA, W.C. and MATSUMOTO, H. (1986).

Visual user-interface of a programming system MOPS². In: THE 1986 IEEE COMPUTER SOCIETY WORKSHOP ON VISUAL LANGUAGES. Dallas, USA, 1986. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 44-53

AHO, A.V., SETHI, R. and ULLMAN, J.D. (1986).

Compilers : principles, techniques and tools. (Wokingham: Addison Wesley).

AHO, A.V. and ULLMAN J.D. (1972).

Introduction to the theory of parsing, translation and compiling, volume 1 - parsing. (Eaglewood Cliffs, N.J., USA: Prentice- Hall).

ALBIZURI-ROMERO, M.B. (1984)

GRASE - A graphical syntax directed editor for structured programming. ACM SIGPLAN Notices 19(2): 28-37.

AMBLER, A.L. and BURNETT, M.M. (1989)

Influence of visual technology on the evolution of language environments. IEEE Computer 31(10): 9-22.

ARTHUR, J.D. and RAGHU, K.S. (1989)

Taskmaster: an interactive, graphical environment for task specification, execution and monitoring. Behaviour and Information Technology 9(3): 219-233.

BACKUS, J. (1977)

Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Communications of the ACM 21(8): 613-641.

BAILEY, M.L., SOCHA, D. and NOTKIN, D. (1988).

Debugging parallel programs using graphical views. In: THE 1988 INTERNATIONAL CONFERENCE OF PARALLEL PROCESSING. Volume II: Software. Pennsylvania, USA. August 15-19, 1988. Proceedings. New York: Institute of Electrical and Electronics Engineers, pp. 46-49.

BARETT, G. (1988).

The semantics and implementation of Occam. DPhil Thesis: Oxford University, Department of Programming Research Group.

BEMMERL, T. (1988).

An integrated and portable tool environment for parallel computers. INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING. Volume II: Software. Pennsylvania, USA. 1988. Proceedings. New York: Institute of Electrical and Electronics Engineers, pp. 50-53.

BHATTACHARYYA, M., COHRS, D. and MILLER, B. (1988)

A visual process connector for Unix. IEEE Software 20(7): pp. 43-50.

BOJANCZYK, A.W. and KIMURA, T.D. (1986).

A systolic parsing algorithm for a visual programming language. In: THE 1986 IEEE FALL JOINT COMPUTER CONFERENCE. 1986. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 48-55.

BRATKO, I. (1986).

Prolog : programming for artificial intelligence. (Wokingham: Addison Wesley).

BROOKS, F.P. (1987)

No silver bullet - essences and accidents of software engineering. IEEE Computer 20(4): 10-19.

BROWN, G.P., CARLING, R.T., HEROT, C.F., KRAMLICH, D.A. and SOUZA, P. (1985)

Program visualisation: graphical support for software development. IEEE Computer, 18(7): pp. 27-35.

BUHR, R.J., KARAM, G.M., HAYES, C.J. and WOODSIDE, C.M. (1989)

Software CAD: a revolutionary approach. IEEE Transactions on Software Engineering 15(3): 235-249.

BURNS, A. (1988).

Programming in Occam 2. (Wokingham: Addison Wesley).

CAVANO, J.P. (1988).

Software issues facing parallel architectures. In: COMPSAC '88 INTERNATIONAL CONFERENCE ON COMPUTER SOFTWARE AND APPLICATIONS. 12TH. Chicago. USA. 1988. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 300-301.

CCITT. (1984).

Functional specification and description language (SDL), Recommendations Z100-104. Plenary Assembly of the CCITT, Malaga-Torremolinos, 1984.

CHANG, S.K., TAUBER, M.J., YU, B. and YU, J.S. (1987).

The SIL-ICON compiler - an icon oriented system generator. In: THE 1987 IEEE WORKSHOP ON LANGUAGES FOR AUTOMATION. 1987. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 17-22.

CHANG, S.K., TORTORA, G., YU, B. and GUERCIO, A. (1987).

Icon purity - towards a formal theory of icons. In: THE 1987 IEEE WORKSHOP ON VISUAL LANGUAGES. Linköping, Sweden. 1987. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 3-16.

CHOI, J.W. and KIMURA, T.D. (1986).

A compiled picture language on the Macintosh. In: THE 1986 ACM SIGSMALL SYMPOSIUM ON SMALL SYSTEMS. San Francisco, December 3-5, 1986. Proceedings. New York: Association of Computing Machinery, pp. 72-80.

CHOW, C. and LAM, S.S. (1988)

PROSPEC : an interactive programming environment for designing and verifying communication protocols. IEEE Transactions on Software Engineering 14 (3) : 327-338.

CLOCKSIN, F.W. and MELLISH, C.S. (1981)

Programming in Prolog. (Springer Verlag : Berlin).

COOK, R.P. and AULETTA, R.J. (1986).

STARLITE : a visual simulation package for software prototyping. In: THE ACM SIGSOFT/SIGPLAN SOFTWARE ENGINEERING SYMPOSIUM. Palo Alto, CA, USA. 1986. Proceedings. New York: Association of Computing Machinery, pp. 102-110.

COX, P.T. and MULLIGAN, I.J. (1985).

Compiling the graphical functional language PROGRAPH. In: THE 1985 ACM SIGSMALL SYMPOSIUM ON SMALL SYSTEMS. Danvers, MA, USA. May 1-3, 1985. Proceedings. New York : Association of Computing Machinery, pp. 34-41.

COX, P.T. and PIETRZYKOWSKI, T. (1985).

Advanced programming aids in PROGRAPH. In: THE 1985 ACM SIGSMALL SYMPOSIUM ON SMALL SYSTEMS. Denver, CO, USA. 1985. Proceedings. New York: Association of Computing Machinery, pp. 27-33.

CROWE, W.D., HASSON, R. and STRAIN-CLARKE, P.E.D. (1989).

A CASE tool for designing deadlock-free Occam programs. In: THE 11TH OCCAM USER GROUP TECHNICAL MEETING. 11TH. Edinburgh, Scotland. 25-26 September, 1989. Proceedings. Amsterdam, Netherlands: IOS, pp. 23-35.

DAHLER, J., GERBER, P., GISIGER, H.P. and KUNDIG. (1988).

A graphical tool for the design and prototyping on distributed systems. In: THE 1988 ZURICH SEMINAR ON DIGITAL COMMUNICATIONS (Mapping New Applications onto New Technology). Zurich. 1988. Proceedings. New York: Institute of Electric and Electronic Engineers, pp. 123-9.

DAVIS, A.L. and KELLER, R.M. (1982)

Data flow program graphs. IEEE Computer 15(2): 26-41.

DELLA-VIGNA, P. and GHEZZI, C. (1978)

Context-free graph grammars. Information and Control 37: 207-233.

DeMARCO, T. and SOCENEANTU, A. (1984).

SYNCHRO: a dataflow command shell for the lilith/modula computer. In: THE 7TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. 7TH. Orlando. FL. USA. 1984. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 207-213.

DEREMER, F. and KRON, H.H. (1976).

Programming-in-the-large versus programming-in-the-small. IEEE Transactions on Software Engineering 2(2): 114-121.

DEREMER, F.L. (1976).

Review of formalisms and notations. In: Lecture notes in computer science 21 - compiler construction - an advanced course second edition, ed. by G Goos and J Hartmanis. (Berlin: Springer- Verlag).

DUTTON, J. (1986).

EDGE - an extended directed graph editor and its applications. In: THE ACM SIGSMALL SYMPOSIUM ON SMALL SYSTEMS. San Francisco. USA. 1986. Proceedings. New York: Association of Computing Machinery, pp. 26-33.

EHRIG, H. (1979).

Introduction to the algebraic theory of graph grammars, In: lecture notes in computer science 73 - Graph Grammars and their application to Computer Science and Biology, ed. by V. Claus, H. Ehrig, and G. Rozenberg. (Berlin: Springer-Verlag).

EHRIG, H., NAGL, M., ROZENBERG, G. and ROSENFELD, A. (1986).

Lecture notes in computer science 291 - graph grammars and their application to computer science. (Berlin: Springer-Verlag).

EISENBACH, S., McLOUGHLIN, L. and SADLER, C. (1989).

Dataflow design as a visual programming language. In: THE 5TH INTERNATIONAL CONFERENCE ON SOFTWARE SPECIFICATION AND DESIGN. 5TH. Pittsburgh. PA. USA. 1989. Proceedings. New York: Association of Computing Machinery, pp. 281-3.

ENGELS, G. (1986).

Graph grammar engineering - a software specification method, In: lecture notes in computer science 291 - graph-grammars and their application to computer science, ed. by H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfield. (Berlin: Springer-Verlag).

FISHER, G. (1988)

An overview of a graphical multilanguage applications environment. IEEE Transactions on Software Engineering 14(6): 774-786.

FOSTER, I. and TAYLOR, S. (1990).

Strand: new concepts in parallel processing. (Eaglewood Cliffs, NJ, USA: Prentice Hall).

GANE, C. and SARSON, T. (1979).

Structured systems analysis: tools and techniques. (Eaglewood Cliffs, NJ, USA: Prentice-Hall International).

GARCIA, M. E. and BERMAN, W.J. (1985).

An approach to concurrent systems debugging. In: THE 5TH INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTER SYSTEMS, 5TH, Denver, Co., USA, May 13-17, 1985. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 507-514.

GIACALONE, A. and SMOLKA, S.A. (1988)

Integrated environments for formally well-founded design and simulation of concurrent systems. IEEE Transactions on Software Engineering 14 (6): 787-801.

GILLET, W.D. and KIMURA, T.D. (1986a).

Parsing two-dimensional languages. In: COMPSAC 1986. THE IEEE COMPUTER SOCIETY INTERNATIONAL COMPUTER SOFTWARE AND HARDWARE APPLICATIONS CONFERENCE, 10TH, Chicago, USA. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 472-477.

GILLET, W.D. and KIMURA, T.D. (1986b).

The syntax and parsing of two-dimensional languages. Technical Report no. WUCS-86-14: Washington University, Saint Louis, Missouri 63130, USA, Department of Computer Science.

GLINERT, E.P. (1986).

Interactive graphical programming environments: six open problems and a possible partial solution. In: COMPSAC 86 IEEE COMPUTER SOCIETY INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 10TH, Chicago, IL, USA, 1986. Proceedings. New York : Insitute of Electrical and Electronic Engineers, pp. 408-410.

GLINERT, E.P. and TANIMOTO, S.L. (1984)

PICT: an interactive graphical programming environment. IEEE Computer 17(11): 7-25.

GOLDSMITH, M. (1990).

Personal communication.

GOTTLER, H. (1989).

Graph grammars, a new paradigm for implementing visual languages. In: ESEC '89. 2ND EUROPEAN SOFTWARE ENGINEERING CONFERENCE. 2ND. Coventry, UK. 1989. Proceedings. Berlin: Springer-Verlag, pp. 336-50.

GRAF, M. (1987).

A visual environment for the design of distributed systems. In: THE 1987 IEEE WORKSHOP ON VISUAL LANGUAGES. Linkoping, Sweden. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 331-345.

GURD, J.R., KIRKHAM, C.C. and BOHM, A.P.W. (1987).

The Manchester dataflow computing system. In: Experimental computing architectures, ed. by J.J. Dongarra. (Amsterdam: North Holland).

HAEBERLI, P.E. (1988)

ConMan : a visual programming language for interactive graphics. Computer Graphics 22 (4) : 103-111.

HARADA M. and KUNII, T.L. (1984).

A recursive graph theory as a formal basis for a visual design language. In: IEEE COMPUTER SOCIETY WORKSHOP ON VISUAL LANGUAGES. Hiroshima. 1989. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 124-133

HAREL, D. (1988)

On visual formalisms. Communications of the ACM 31(5): 514-529.

HAREL, D., PNUELI, A., SCHMIDT, P. and SHERMAN, R. (1987).

On the formal semantics of statecharts. In: THE SECOND IEEE SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE. 2ND. 1987. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 54-64.

HARTER, P. K., HEIMBIGNER, D. M. and KING, R. (1985).

IDD: an interactive distributed debugger. In: THE INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTER SYSTEMS (IEEE). 5TH. Denver, Co., USA, May 13-17, 1985. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp.498-506.

HEKMATPOUR, S. and WOODMAN, M. (1987).

Formal specification of graphical notations and graphical software tools. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE. (1ST). Strasbourg, France, 1987. Proceedings. Springer-Verlag, pp. 297-305.

HOARE, C.A.R. (1985).

Communicating Sequential Processes. (Hemel Hempstead: Prentice Hall International Series in Computer Science).

HUGHES, C. (1989)

Future perfect - the alvey parallel simulation facility project. Parallelogram 1(17): 16-18.

HUTCHINGS, A.M.J. (1986).

Edinburgh prolog (the new implementation user's manual). (University of Edinburgh, Scotland: AI Applications Institute).

IANUCHI, R.A. (1983).

A critique of multiprocessing Von Neumann style. In: THE TENTH ANNUAL SYMPOSIUM ON COMPUTER ARCHITECTURE. 10TH. Stockholm, 1983. Proceedings. New York: Association of Computing Machinery, pp. 426-436.

INMAN, E. (1987).

A syntax-directed graphics editor. In: Human-computer interaction, ed. by H.J. Bullinger and B. Shackel. (N. Ireland: Elsevier Science Publications).

INMOS. (1984).

Occam programming manual. (Hemel Hempstead, U.K.: Prentice Hall International).

INMOS. (1988).

Transputer development system. (Hemel Hempstead, U.K.: Prentice Hall International).

INMOS. (1989a).

The transputer development and IQ systems databook. (Bristol, U.K.: Inmos Ltd).

INMOS. (1989b).

The transputer applications notebook - systems and performance. (Bristol, U.K.: Inmos Ltd).

JONES, C.B. (1990).

Systematic software development using VDM (Second Edition). (Hemel Hempstead: Prentice Hall International series in computer science).

JONES, G. and GOLDSMITH, M. (1988)

Programming in Occam 2. (Hemel Hempstead: Prentice Hall International).

JOYCE, J. and UNGER, B. (1985).

Graphical monitoring of distributed systems. In: THE SCS CONFERENCE ON ARTIFICIAL INTELLIGENCE, GRAPHICS AND SIMULATION, San Diego, Ca., USA, Jan 1985. Proceedings. New York: Society for Computer Simulation, pp. 85-92.

JOYCE, J., LOMOW, G., SLIND, K. and UNGER, B. (1987).

Monitoring distributed systems. ACM Transactions on Computer Systems 5(2): 121-150.

KAPLAN, S.M., GOERING, S.K. and CAMPBELL, R.H. (1989).

Specifying concurrent systems with delta-grammars. In: INTERNATIONAL WORKSHOP ON SOFTWARE SPECIFICATION AND DESIGN, 5TH, 1989. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 20-27.

KAUL, M. (1982).

Parsing of graphs and their applications in computer science. In: Lecture notes in computer science, Vol 153, ed. by H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfield. (Berlin: Springer-Verlag).

KERNER, H. and RAINEL, H. (1986).

EDDA: a language based on petrinets and the dataflow principle for the development of parallel programs. Microprocessing and Microprogramming (18): 299-306.

KERRIDGE, J. (1987).

Occam programming: a practical approach. (Oxford: Blackwell Scientific Publishers).

KERNIGHAN, B.W. and RITCHIE, D.M. (1988).

The C programming language, second edition. (Eaglewood Cliffs, NJ, USA: Prentice-Hall).

KIMURA, T.D. (1986).

COMPSAC 86 panel session on visual programming : possibilities and limitations. IEEE COMPSAC INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 10TH, Chicago, IL, USA, 1986. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 404-411.

KIMURA, T.D. (1988).

Visual programming by transaction network. In: THE 21ST ANNUAL HAWAII INTERNATIONAL CONFERENCE ON SYSTEMS SCIENCE VOLUME II: SOFTWARE, 21ST, Hawaii, 1988. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 648-54.

KOKEN, D., TEITELBAUM, T., CHEN, W., FIELD, J., PUGH, W. and ZANDER, B.V. (1987).

ALEX - an alexical programming language. In: THE 1987 IEEE WORKSHOP ON VISUAL LANGUAGES. Linköping, Sweden. 1987. Proceedings. New York: Institute of Electrical and Electronics Engineers, pp.315-329.

KOYAMADA, M. and SHIGO, O. (1984).

Design support facilities for switching systems software. In: THE 1984 IEEE COMPUTER SOCIETY WORKSHOP ON VISUAL LANGUAGES. Hiroshima. 1984. Proceedings. New York: Institute of Electrical and Electronics Engineers, pp.47-52.

KRAMER, J., MAGEE, J. and NG, K. (1989).

Graphical configuration programming. IEEE Computer 31(10): 53-65.

LAKIN, F. (1987).

Visual grammars for visual languages. In: AMERICAN ASSOCIATION FOR ARTIFICIAL INTELLIGENCE CONFERENCE. 1987. Proceedings. New York: American Association for Artificial Intelligence, pp. 683- 688.

LEBLANC, T.J. and MILLER, B.P. (1988).

PROCEEDINGS OF THE WORKSHOP ON PARALLEL AND DISTRIBUTED DEBUGGING. May 5-6, 1988. Published as SIGPLAN notices 24(1), New York: Association for Computer Machinery.

LEHR, T., SEGALL, Z., VRSALOVIV, D.F., CAPLAN, E., CHUNG, A.L. and FINEMAN, C.E. (1989)

Visualising performance debugging. IEEE Computer 31(10): 38-51.

LEIB, S. (1990).

The ultimate interface (virtual worlds technology). Information WEEK (USA) 276 June 1990 : 46-48.

LEVCO, (1990).

Express System. Product Information. Levco Inc., Del Mar, Ca., USA.

LODDING, K.N. (1983)

Iconic interfacing. IEEE Computer Graphics and Applications 3(2): 11-20.

LOONEY, M.J. (1988).

Mascot design support environment. In: MILLCOMP '88: MILITARY COMPUTER GRAPHICS AND SOFTWARE CONFERENCE. London. 1988. Proceedings. (Tunbridge Wells.: Microwave Exhibitions and Publishers Ltd), pp. 216-224.

LUDOLPH, F., CHOW, Y., INGALLS, D., WALLACE, S. and DOYLE, S. (1988).
The fabrik programming environment. In: IEEE WORKSHOP ON VISUAL LANGUAGES, Pittsburgh, USA, 1988. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 222-230.

MAY, D. (1987).
Occam2 language definition. (Bristol, U.K.: Inmos Limited).

McDOWELL, C.E. (1988).
Viewing anomalous states in parallel programs. In: THE 1988 INTERNATIONAL CONFERENCE OF PARALLEL PROCESSING, Volume II: Software, Pennsylvania, USA, August 15-19, 1988. Proceedings. New York: Institute of Electrical and Electronics Engineers, pp. 54-57.

MILNER, R. (1980).
A calculus of communicating systems, lecture notes in computer science 92. (New York: Springer-Verlag).

MONDEN, N., YOSHINO, Y., HIRAKAWA, M., TANAKA, M. and ICHIKAWA, T. (1984).
HI-VISUAL : a language supporting visual interaction in programming. In: IEEE COMPUTER SCIENCE WORKSHOP ON VISUAL LANGUAGES, Hiroshima, December 6-8, 1984. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 199-205.

MORAN, M. and FELDMAN, M.B. (1985).
Towards graphical animated debugging of concurrent programs in ADA. In: THE INTERNATIONAL SYMPOSIUM ON NEW DIRECTIONS IN COMPUTING, August 12-14, 1985. Proceedings. New York: Electrical and Electronics Engineers, pp.344-351.

MOURLIN, F. and Cournarie, E. (1990)
A graphical environment for Occam programming. In: APPLICATIONS OF TRANSPUTERS 1 - INTERNATIONAL CONFERENCE. (1ST), 1990. Proceedings. IOS: Amsterdam, pp. 252-61.

MYERS, B.A. (1988).
The state of the art in visual programming and program visualisation. Technical Report number CMU-CS-88-114. Carnegie Mellon University, Pittsburgh, PA, USA, Computer Science Department. Presented at British Computer Society Displays Group's symposium on visual programming and program visualisation, London, 16 March 1988.

NAGL, M. (1986a).
Set theoretic approaches to graph grammars. In: Lecture notes in computer science 291 - graph-grammars and their application to computer science, ed. by H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfield. (Berlin: Springer-Verlag).

NAGL, M. (1986b).

A software development environment based on graph technology. In: Lecture notes in computer science 291 - graph-grammars and their application to computer science, ed. by H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfield. (Berlin: Springer-Verlag).

NAKAMURA, K., FUJIMOTO, H., SUZUKI, T., TARUI, Y. and KIYOKANE, Y. (1986).

Visual programming environment in communications software. In: GLOBE COM'86 - THE IEEE TELECOMMUNICATIONS CONFERENCE. 1986. Proceedings. New York: Institute of Electrical and Electronics Engineers, pp. 435-9.

NICHOLS, K.M. and EDMARK, J.T. (1988)

Modelling multicomputer systems with PARET. IEEE Computer 21(5): 39-48.

ORR, R.A., NORRIS, M.T., TINKER, R. and ROUCH, C.D.V. (1988).

Tools for real time system design. In: THE 10TH INTERNATIONAL CONFERENCE SOFTWARE ENGINEERING (IEEE/ACM). 10TH. Singapore, 1988. Proceedings. New York: Institute of Electrical and Electronics Engineers/Association of Computing Machinery pp.130- 137.

PANCAKE, C.M. and UTTER, S. (1989).

Models for visualisation in parallel debuggers. In: SUPERCOMPUTER '89. Mannheim, Germany, 1989. Proceedings. New York: Association of Computing Machinery, pp. 627-36.

PERIHELION, (1988).

The Helios user's manual. (Somerset, U.K.: Perihelion Software Ltd).

PETERSON, J.L. (1977)

Petri nets. ACM Computing Surveys. Volume SE-6: 440-449, September, 1977.

PETERSON, J.L. (1980)

A note on coloured petri-nets. Information Processing Letters 11 (1): 40-43.

PONG, M.C. (1986).

A graphical language for concurrent programming. In: THE 1986 IEEE COMPUTER SOCIETY WORKSHOP ON VISUAL LANGUAGES. Dallas, Texas, 1986. Proceedings.. New York: Institute of Electric and Electronic Engineers, pp. 26-33.

PRATT, T.W. (1971)

Pair grammars, graph languages and string to graph translations. Journal of Systems Science 5: 560-595.

RAEDER, G. (1985)

A survey of current graphical programming techniques. IEEE Computer 18(8): 11-25.

ROBERTS, M. (1990a)

A new specialist interest group - graphical program development tools. Occam User Group Newsletter 12: 21-22

ROBERTS, M. (1990b)

Graphical program development tools specialist interest group report. Occam User Group Newsletter 13 :41-42.

ROBERTS, M. and SAMWELL, P. (1989).

A visual programming system for the development of parallel software. In: SOFTWARE ENGINEERING FOR REAL TIME SYSTEMS. 2ND. Cirencester, UK, 1989. Proceedings. London: Institute of Electrical Engineers, pp. 75-79.

ROBERTS, M. and SAMWELL, P. (1990).

A visual programming system for the transputer. In: CONFERENCE OF NORTH AMERICAN TRANSPUTER USERS GROUP, 3RD, Sunnyvale, CA, USA, April 26-27, 1990. Proceedings. IOS Press: Washington, pp. 235-244.

ROSCOE, A.W. (1984).

A denotational semantics for Occam. In: THE PITTSBURGH SEMINAR ON CONCURRENCY. LECTURE NOTES IN COMPUTER SCIENCE 197. Pittsburgh, 1984. Proceedings. Berlin: Springer-Verlag, pp. 306-329.

ROSCOE, A.W. and HOARE, C.A.R. (1986).

The laws of Occam programming technical monograph PRG-53. Technical Report: Oxford University Computing Laboratory, Oxford, U.K.: Programming Research Group.

ROSENBERG, G. (1986).

An introduction to the NLC way of rewriting graphs. In: Lecture notes in computer science 291 - graph-grammars and their application to computer science: ed. by H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfield. (Berlin: Springer-Verlag).

SCHWAN, K. and MATTHEWS, J. (1986)

Graphical views of parallel programs. ACM SIGSOFT Software Engineering Notes 11(3): 51-64.

SHATZ, S.M. and WANG, J. (1989).

Tutorial: distributed software engineering (IEEE Cat No. EH0283-2). (New York: Institute of Electrical and Electronic Engineers).

SHU, N. C. (1988).

Visual Programming. (New York: VanNostrand-Reinhold).

SIMPSON, H. (1986)

The mascot method. Software Engineering Journal, May, 1986: 103- 120.

SNYDER, L. (1984)

Parallel programming and the poker programming environment. IEEE Computer 17 (7): 27-36.

SNYDER, L. (1987).

Hearts: a dialect of the poker programming environment specialised to systolic computation. In: Systolic Arrays. Ed by W. Moore, A. McCabe, and R. Urquhart. (Bristol: Adam Hilger), pp. 71-80.

SPIVEY, J.M. (1989).

The Znotation: a reference manual. (Hemel Hempstead: Prentice Hall).

SOBEK, S., AZAM, M. and BROWNE, J.C. (1988).

Architecture and language independent parallel programming: a feasibility demonstration. In: THE 1988 INTERNATIONAL CONFERENCE OF PARALLEL PROCESSING, Volume II: Software, Pennsylvania, USA, August 15-19, 1988. Proceedings. New York: Institute of Electrical and Electronics Engineers, pp. 80-83.

STEPHENSON, M. and BOUDILLET, O. (1988).

GECKO: a graphical tool for the modelling and manipulation of Occam software and hardware topologies. In: OCCAM AND THE TRANS PUTER - RESEARCH AND APPLICATIONS, OCCAM USER GROUP TECHNICAL MEETING, 9TH, Southampton, U.K. 1988. Proceedings. Amsterdam: IOS, pp. 139-144.

STEPNEY, S. (1987).

GRAIL: graphical representation of activity, interconnection and loading. In: THE 7TH OCCAM USER GROUP TECHNICAL MEETING, 7TH, Grenoble, France, 1987. Proceedings. Amsterdam: IOS, (published without pagination).

STONE, J.M. (1989)

A graphical representation of concurrent processes. SIGPLAN Notices 24(1): 226-35.

STOTTS, P.D. (1988).

The PFG language - visual programming for concurrent computation. In: THE 1988 INTERNATIONAL CONFERENCE OF PARALLEL PROCESSING, Volume II: Software, Pennsylvania, USA, August 15-19, 1988. Proceedings. New York: Institute of Electrical and Electronics Engineers, pp. 72-79.

STOVSKY, M.P. and WEIDE, B.W. (1987).

STILE: a graphical design and development environment. In: DIGEST COMPCON 1987, CS PRESS, Los Alamitos, CA, USA. pp. 247-250.

SUN, (1989)

Sunview 1 programmer's guide, 1989. (Sun Microsystems Inc., Mountain View, CA, USA).

SZWILLUS, G. (1987).

GECS - a system for generating graphical editors. In: THE HUMAN COMPUTER INTERACTION INTERACT'87 CONFERENCE. 2ND. Stuttgart, Germany. 1987. Proceedings. North Holland: Elsevier Science Publishers, pp. 135-141.

TANG, H.W.H. (1987).

Monitoring tools for parallel systems. Technical Report No. UMCS-87-12-3: University of Manchester, Department of Computer Science.

TILLEY, M.S. (1990).

Personal communication.

TOMBOULIAN, S., CROCKETT, T.W. and MIDDLETON, D. (1988).

A visual programming environment for the navier-stokes computer. In: THE 1988 INTERNATIONAL CONFERENCE OF PARALLEL PROCESSING, Volume II: Software. Pennsylvania, USA. August 15-19, 1988. Proceedings. New York: Institute of Electrical and Electronics Engineers, pp. 67-71.

TRANSTECH (1989).

NTP 1000 technical reference manual. Transtect Ltd : High Wycomb, U.K.

VORNBERGER, O. and ZEPPENFELD, K. (1990).

Graphical visualisation of distributed algorithms. In: CONFERENCE OF THE NORTH AMERICAN TRANSPUTER USERS GROUP, 3RD. Sunnyvale, CA, USA. April 26-27, 1990. Proceedings. IOS: Amsterdam.

WEST, A.J. (1987).

Monitoring distributed occam programs. Thesis for the degree of Ph.D. in faculty of science: University of Manchester, Department of Computer Science.

WEST, S. and CAPON, P. (1990).

A high level software environment for transputer based systems. In: THE 12TH OCCAM USER GROUP TECHNICAL MEETING, 12TH. Exeter, U.K., 1990. Proceedings. Amsterdam: IOS, pp. 232-242.

WINTERBOTTOM, P., and OSMON, P. (1990).

Topsy: an extensible Unix multicomputer. In: UK IT CONFERENCE, publication no 316. Proceedings. London: Institute of Electrical Engineers, pp. 164-76..

YAU, S.S. and GRABOW, P.C. (1981)

A model for representing programs using hierarchical graphs. IEEE Transactions on Software Engineering SE-7(6): 556-571.

YOSHIMOTO, I., MONDEN, N., HIRAKAWA, M., TANAKA, M. and ICHIKAWA, T. (1986).

Interactive iconic programming facility in HI-VISUAL. In: THE IEEE COMPUTER SOCIETY WORKSHOP ON VISUAL LANGUAGES, Dallas, 1986. Proceedings. New York: Institute of Electrical and Electronic Engineers, pp. 34-41.

ZIMMERMAN, M., PERRENOUD, F. and SCHIPER, A. (1988)

Understanding concurrent programming through program animation. ACM SIGCHE Bulliten 20(1): 27-31.